

Netty®

权威指南

李林锋 / 著

Java高性能NIO通信首选框架

大数据时代构建高可用分布式系统利器

Netty: The Definitive Guide

内容简介

《Netty权威指南》是异步非阻塞通信领域的经典之作，基于最新版本Netty 5.0编写，是国内首本深入介绍Netty原理和架构的技术书籍，也是作者多年实战经验的总结和浓缩。内容包含基础功能、高级应用、系统架构、源码分析和行业应用，深入阐述了Java I/O的Netty NIO开发、Netty编解码开发、Netty多协议开发等各方面的技术要点，包含了对源码的深刻解读，并且对Netty的应用现状和未来趋势进行分析，旨在帮助从业人员提升自我，更快更明确地发展职业道路。

本书适合架构师、设计师、软件开发工程师、测试人员和其他对Java NIO框架、Java通信感兴趣的相关人士阅读，相信通过学习本书，能够熟悉和掌握Netty这一优秀的异步通信框架，实现高可用分布式系统的构建。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Netty权威指南 / 李林锋著. —北京：电子工业出版社，2014.6

ISBN 978-7-121-23343-2

I. ①N... II. ①李... III. ①JAVA语言—程序设计—指南 IV.

①TP312-62

中国版本图书馆CIP数据核字（2014）第114646号

策划编辑：孙学瑛

责任编辑：徐津平

印 刷：北京中新伟业印刷有限公司

装 订：河北省三河市路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编100036

开 本：787×980 1/16 印张：32.75 字数：625千字

版 次：2014年6月第1版

印 次：2014年6月第1次印刷

定 价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

大约在2008年的时候，我参与设计和开发的一个电信系统在月初出帐期，总是发生大量的连接超时和读写超时异常，业务的失败率相比于平时高了很多，报表中的很多指标都差强人意。后来经过排查，发现问题的主要原因出现在下游网元的处理性能上，月初的时候BSS出帐，在出帐期间BSS系统运行缓慢，由于双方采用了同步阻塞式的HTTP+XML进行通信，导致任何一方处理缓慢都会影响对方的处理性能。按照故障隔离的设计原则，对方处理速度慢或者不回应答，不应该影响系统的其他功能模块或者协议栈，但是在同步阻塞I/O通信模型下，这种故障传播和相互影响是不可避免的，很难通过业务层面解决。

受限于当时Tomcat和Servlet的同步阻塞I/O模型，以及在Java领域异步HTTP协议栈的技术积累不足，当时我们并没有办法完全解决这个问题，只能通过调整线程池策略和HTTP超时时间来从业务层面做规避。

2009年，由于对技术的热爱，我作为业务骨干被领导派去参加一个重点业务平台的研发工作，与两位资深的架构师（其中一位工作20年，做华为交换机出身）共同参与。这是我第一次全面接触异步I/O编程和高性能电信级协议栈的开发，眼界大开——异步高性能内部协议栈、异步HTTP、异步SOAP、异步SMPP.....所有的协议栈都是异步非阻塞。后来的性能测试表明：基于Reactor模型统一调度的长连接和短连接协议栈，无论是性能、可靠性还是可维护性，都可以“秒杀”传统基于BIO开发的应用服务器和各种协议栈，这种差异本质上是一种代差。

在我从事异步NIO编程的2009年，业界还没有成熟的NIO框架，那个时候Mina刚刚开始起步，功能和性能都达不到商用标准。最困难的是，国内Java领域的异步通信还没有流行，整个业界的积累都非常少。那个时候资料匮乏，能够交流和探讨的圈内人很少，一旦踩住“地雷”，就需要夜以继日地维护。在随后2年多的时间里，经历了10多次的在通宵、凌晨被一线的运维人员电话吵醒等种种磨难之后，我们自研的NIO框架才逐渐稳定和成熟。期间，解决的BUG总计20~30个。

从2004年JDK1.4首次提供NIO 1.0类库到现在，已经过去了整整10年。JSR 51的设计初衷就是让Java能够提供非阻塞、具有弹性伸缩能力的异步I/O类库，从而结束Java在高性能服务器领域的不利地位。然而，在相当长的一段时间里，Java的NIO编程并没有流行起来，究其原因如下。

1. 大多数高性能服务器，被C和C++语言盘踞，由于它们可以直接使用操作系统的异步I/O能力，所以对JDK的NIO并不关心；
2. 移动互联网尚未兴起，基于Java的大规模分布式系统极少，很多中小型应用服务对于异步I/O的诉求不是很强烈；
3. 高性能、高可靠性领域，例如银行、证券、电信等依然以C++为主导，Java充当打杂的角色，NIO暂时没有用武之地；
4. 当时主流的J2EE服务器，几乎全部基于同步阻塞I/O构建，例如Servlet、Tomcat等，由于它们应用广泛，如果这些容器不支持NIO，用户很难具备独立构建异步协议栈的能力；
5. 异步NIO编程门槛比较高，开发和维护一款基于NIO的协议栈对很多中小型公司来说像是一场噩梦；

6. 业界NIO框架不成熟，很难商用；

7. 国内研发界对NIO的陌生和认识不足，没有充分重视。

基于上述几种原因，NIO编程的推广和发展长期滞后。值得欣慰的是，随着大规模分布式系统、大数据和流式计算框架的兴起，基于Java来构建这些系统已经成为主流，NIO编程和NIO框架在此期间得到了大规模的商用。在互联网领域，阿里的分布式服务框架Dubbo、RocketMQ，大数据的基础序列化和通信框架Avro，以及很多开源的软件都已经开始使用Netty来构建高性能、分布式通信能力，Netty社区的活跃度也名列前茅。根据目前的信息，Netty已经在如下几个领域得到了大规模的商业应用。

1. 互联网领域；

2. 电信领域；

3. 大数据领域；

4. 银行、证券等金融领域；

5. 游戏行业；

6. 电力等企业市场。

2014年春节前，我分享了一篇博文《Netty5.0架构剖析和源码解读》，短短1个月下载量达到了4000多。很多网友向我咨询NIO编程技术、NIO框架如何选择等问题，也有一些圈内朋友和出版社邀请我写一本关于Netty的技术书籍。作为最流行、表现最优异的NIO框架，Netty深受大家喜爱，但是长期以来除了UserGuide之外，国内鲜有Netty相关

的技术书籍供广大NIO编程爱好者学习和参考。由于Netty源码的复杂性和NIO编程本身的技术门槛限制，对于大多数读者而言，通过自己阅读和分析源码来深入掌握Netty的设计原理和实现细节是件困难的事情。从2011年开始我系统性的分析和应用了Netty和Mina，转瞬间已经过去了3年多。在这3年的时间里，我们的系统经受了无数严苛的考验，在这个过程中，对Netty和Mina有了更深刻的体验，也积累了丰富的运维和实战经验。我们都是开源框架Netty的受益者，为了让更多的朋友和同行能够了解NIO编程，深入学习和掌握Netty这个NIO利器，我打算将我的经验和大家分享，同时也结束国内尚无Netty学习教材的尴尬境地。

联系方式

尽管我也有技术洁癖，希望诸事完美，但是由于Netty代码的庞杂和涉及到的知识点太多，一本书籍很难涵盖所有的功能点。如有遗漏或者错误，恳请大家能够及时批评和指正，如果你有好的建议或者想法，也可以联系我。我的联系方式如下。

邮箱：neu_lilinfeng@sina.com。

新浪微博：Nettying。

微信：Nettying。

致谢

如果说个人能够改变自己命运的话，对于程序员来说，唯有通过不断的学习和实践，努力提升自己的技能，才有可能找到更好的机会，充分发挥和体现自己的价值。我希望本书能够为你的成功助一臂之力。

感谢博文视点的策划编辑丁一琼和幕后的美编，正是你们的辛苦工作才保证了本书能够顺利出版；感谢华为Netty爱好者和关注本书的领导同事们的支持，你们的理解和鼓励为我提供了足够的勇气。感谢我的家人和老婆的支持，写书占用了我几乎所有的业余时间，没有你们的理解和支持，我很难安心写作。

最后感谢Netty中国社区的朋友，我的微博粉丝和所有喜欢Netty的朋友们，你们对技术的热情是鼓励我写书的最重要动力，没有你们，就没有本书。希望大家一如既往的喜欢NIO编程，喜欢Netty，以及相互交流和分享，共同推动整个国内异步高性能通信领域的技术发展。

李林锋

5月11日于南京紫轩阁

目 录

[前言](#)

[基础篇 走进Java NIO](#)

[第1章 Java的I/O演进之路](#)

[1.1 I/O基础入门](#)

[1.1.1 Linux网络I/O模型简介](#)

[1.1.2 I/O多路复用技术](#)

[1.2 Java的I/O演进](#)

[1.3 总结](#)

[第2章 NIO入门](#)

[2.1 传统的 BIO 编程](#)

[2.1.1 BIO通信模型图](#)

[2.1.2 同步阻塞式I/O创建的TimeServer源码分析](#)

[2.1.3 同步阻塞式I/O创建的TimeClient源码分析](#)

[2.2 伪异步I/O编程](#)

[2.2.1 伪异步I/O模型图](#)

[2.2.2 伪异步式I/O创建的TimeServer源码分析](#)

[2.2.3 伪异步I/O弊端分析](#)

[2.3 NIO编程](#)

[2.3.1 NIO类库简介](#)

[2.3.2 NIO服务端序列图](#)

[2.3.3 NIO创建的TimeServer源码分析](#)

[2.3.4 NIO客户端序列图](#)

[2.3.5 NIO创建的TimeClient源码分析](#)

[2.4 AIO编程](#)

[2.4.1 AIO创建的TimeServer源码分析](#)

[2.4.2 AIO创建的TimeClient源码分析](#)

[2.4.3 AIO版本时间服务器运行结果](#)

[2.5 4种I/O的对比](#)

[2.5.1 概念澄清](#)

[2.5.2 不同I/O模型对比](#)

[2.6 选择Netty的理由](#)

[2.6.1 不选择Java原生NIO编程的原因](#)

[2.6.2 为什么选择Netty](#)

[2.7 总结](#)

[入门篇 Netty NIO开发指南](#)

[第3章 Netty入门应用](#)

[3.1 Netty开发环境的搭建](#)

[3.1.1 下载Netty的软件包](#)

[3.1.2 搭建Netty应用工程](#)

[3.2 Netty服务端开发](#)

[3.3 Netty客户端开发](#)

[3.4 运行和调试](#)

[3.4.1 服务端和客户端的运行](#)

[3.4.2 打包和部署](#)

[3.5 总结](#)

[第4章 TCP粘包/拆包问题的解决之道](#)

[4.1 TCP粘包/拆包](#)

[4.1.1 TCP粘包/拆包问题说明](#)

[4.1.2 TCP粘包/拆包发生的原因](#)

[4.1.3 粘包问题的解决策略](#)

[4.2 未考虑TCP粘包导致功能异常案例](#)

[4.2.1 TimeServer的改造](#)

[4.2.2 TimeClient的改造](#)

[4.2.3 运行结果](#)

[4.3 利用LineBasedFrameDecoder解决TCP粘包问题](#)

[4.3.1 支持TCP粘包的TimeServer](#)

[4.3.2 支持TCP粘包的TimeClient](#)

[4.3.3 运行支持TCP粘包的时间服务器程序](#)

[4.3.4 LineBasedFrameDecoder和StringDecoder的原理分析](#)

[4.4 总结](#)

[第5章 分隔符和定长解码器的应用](#)

[5.1 DelimiterBasedFrameDecoder应用开发](#)

[5.1.1 DelimiterBasedFrameDecoder服务端开发](#)

[5.1.2 DelimiterBasedFrameDecoder客户端开发](#)

[5.1.3 运行DelimiterBasedFrameDecoder服务端和客户端](#)

[5.2 FixedLengthFrameDecoder应用开发](#)

[5.2.1 FixedLengthFrameDecoder服务端开发](#)

[5.2.2 利用telnet命令行测试EchoServer服务端](#)

[5.3 总结](#)

[中级篇 Netty编解码开发指南](#)

[第6章 编解码技术](#)

[6.1 Java序列化的缺点](#)

[6.1.1 无法跨语言](#)

[6.1.2 序列化后的码流太大](#)

[6.1.3 序列化性能太低](#)

[6.2 业界主流的编解码框架](#)

[6.2.1 Google的Protobuf介绍](#)

[6.2.2 Facebook的Thrift介绍](#)

[6.2.3 JBoss Marshalling介绍](#)

[6.3 总结](#)

[第7章 Java序列化](#)

[7.1 Netty Java序列化服务端开发](#)

[7.2 Java序列化Netty客户端开发](#)

[7.3 运行结果](#)

[7.4 总结](#)

[第8章 Google Protobuf编解码](#)

[8.1 Protobuf的入门](#)

[8.1.1 Protobuf开发环境搭建](#)

[8.1.2 Protobuf编解码开发](#)

[8.1.3 运行Protobuf例程](#)

[8.2 Netty的Protobuf服务端开发](#)

[8.2.1 Protobuf版本的图书订购服务端开发](#)

[8.2.2 Protobuf版本的图书订购客户端开发](#)

[8.2.3 Protobuf版本的图书订购程序功能测试](#)

[8.3 Protobuf的使用注意事项](#)

[8.4 总结](#)

[第9章 JBoss Marshalling编解码](#)

[9.1 Marshalling开发环境准备](#)

[9.2 Netty的Marshalling服务端开发](#)

[9.3 Netty的Marshalling客户端开发](#)

[9.4 运行Marshalling客户端和服务端例程](#)

[9.5 总结](#)

[高级篇 Netty多协议开发和应用](#)

[第10章 HTTP协议开发应用](#)

[10.1 HTTP协议介绍](#)

[10.1.1 HTTP协议的URL](#)

[10.1.2 HTTP请求消息（HttpRequest）](#)

[10.1.3 HTTP响应消息（HttpResponse）](#)

[10.2 Netty HTTP服务端入门开发](#)

[10.2.1 HTTP服务端例程场景描述](#)

[10.2.2 HTTP服务端开发](#)

[10.2.3 Netty HTTP文件服务器例程运行结果](#)

[10.3 Netty HTTP+XML协议栈开发](#)

[10.3.1 开发场景介绍](#)

[10.3.2 HTTP+XML协议栈设计](#)

[10.3.3 高效的XML绑定框架JiBx](#)

[10.3.4 HTTP+XML编解码框架开发](#)

[10.3.5 HTTP+XML协议栈测试](#)

[10.3.6 小结](#)

[10.4 总结](#)

[第11章 WebSocket协议开发](#)

[11.1 HTTP协议的弊端](#)

[11.2 WebSocket入门](#)

[11.2.1 WebSocket背景](#)

[11.2.2 WebSocket连接建立](#)

[11.2.3 WebSocket生命周期](#)

[11.2.4 WebSocket连接关闭](#)

[11.3 Netty WebSocket协议开发](#)

[11.3.1 WebSocket服务端功能介绍](#)

[11.3.2 WebSocket服务端开发](#)

[11.3.3 运行WebSocket服务端](#)

[11.4 总结](#)

[第12章 UDP协议开发](#)

[12.1 UDP协议简介](#)

[12.2 UDP服务端开发](#)

[12.3 UDP客户端开发](#)

[12.4 运行UDP例程](#)

[12.5 总结](#)

[第13章 文件传输](#)

[13.1 文件的基础知识](#)

[13.1.1 文件的概念](#)

[13.1.2 文件路径](#)

[13.1.3 文件名称](#)

[13.1.4 FileChannel简介](#)

[13.2 Netty文件传输开发](#)

[13.3 运行Netty文件传输服务例程](#)

[13.4 总结](#)

[第14章 私有协议栈开发](#)

[14.1 私有协议介绍](#)

[14.2 Netty协议栈功能设计](#)

[14.2.1 网络拓扑图](#)

[14.2.2 协议栈功能描述](#)

[14.2.3 通信模型](#)

[14.2.4 消息定义](#)

[14.2.5 Netty协议支持的字段类型](#)

[14.2.6 Netty协议的编解码规范](#)

[14.2.7 链路的建立](#)

[14.2.8 链路的关闭](#)

[14.2.9 可靠性设计](#)

[14.2.10 安全性设计](#)

[14.2.11 可扩展性设计](#)

[14.3 Netty协议栈开发](#)

[14.3.1 数据结构定义](#)

[14.3.2 消息编解码](#)

[14.3.3 握手和安全认证](#)

[14.3.4 心跳检测机制](#)

[14.3.5 断连重连](#)

[14.3.6 客户端代码](#)

[14.3.7 服务端代码](#)

[14.4 运行协议栈](#)

[14.4.1 正常场景](#)

[14.4.2 异常场景：服务端宕机重启](#)

[14.4.3 异常场景：客户端宕机重启](#)

[14.5 总结](#)

[源码分析篇 Netty功能介绍和源码分析](#)

[第15章 ByteBuf和相关辅助类](#)

[15.1 ByteBuf功能说明](#)

[15.1.1 ByteBuf的工作原理](#)

[15.1.2 ByteBuf的功能介绍](#)

[15.2 ByteBuf源码分析](#)

[15.2.1 ByteBuf的主要类继承关系](#)

[15.2.2 AbstractByteBuf源码分析](#)

[15.2.3 AbstractReferenceCountedByteBuf源码分析](#)

[15.2.4 UnpooledHeapByteBuf源码分析](#)

[15.2.5 PooledByteBuf内存池原理分析](#)

[15.2.6 PooledDirectByteBuf源码分析](#)

[15.3 ByteBuf相关的辅助类功能介绍](#)

[15.3.1 ByteBufHolder](#)

[15.3.2 ByteBufAllocator](#)

[15.3.3 CompositeByteBuf](#)

[15.3.4 ByteBufUtil](#)

[15.4 总结](#)

[第16章 Channel和Unsafe](#)

[16.1 Channel 功能说明](#)

[16.1.1 Channel的工作原理](#)

[16.1.2 Channel的功能介绍](#)

[16.2 Channel源码分析](#)

[16.2.1 Channel的主要继承关系类图](#)

[16.2.2 AbstractChannel源码分析](#)

[16.2.3 AbstractNioChannel源码分析](#)

[16.2.4 AbstractNioByteChannel源码分析](#)

[16.2.5 AbstractNioMessageChannel源码分析](#)

[16.2.6 AbstractNioMessageServerChannel源码分析](#)

[16.2.7 NioServerSocketChannel源码分析](#)

[16.2.8 NioSocketChannel源码分析](#)

[16.3 Unsafe功能说明](#)

[16.4 Unsafe源码分析](#)

[16.4.1 Unsafe继承关系类图](#)

[16.4.2 AbstractUnsafe源码分析](#)

[16.4.3 AbstractNioUnsafe源码分析](#)

[16.4.4 NioByteUnsafe源码分析](#)

[16.5 总结](#)

[第17章 ChannelPipeline和ChannelHandler](#)

- [17.1 ChannelPipeline功能说明](#)
 - [17.1.1 ChannelPipeline的事件处理](#)
 - [17.1.2 自定义拦截器](#)
 - [17.1.3 构建pipeline](#)
 - [17.1.4 ChannelPipeline的主要特性](#)
- [17.2 ChannelPipeline源码分析](#)
 - [17.2.1 ChannelPipeline的类继承关系图](#)
 - [17.2.2 ChannelPipeline对ChannelHandler的管理](#)
 - [17.2.3 ChannelPipeline的inbound事件](#)
 - [17.2.4 ChannelPipeline的outbound事件](#)
- [17.3 ChannelHandler功能说明](#)
 - [17.3.1 ChannelHandlerAdapter功能说明](#)
 - [17.3.2 ByteToMessageDecoder功能说明](#)
 - [17.3.3 MessageToMessageDecoder功能说明](#)
 - [17.3.4 LengthFieldBasedFrameDecoder功能说明](#)
 - [17.3.5 MessageToByteEncoder功能说明](#)
 - [17.3.6 MessageToMessageEncoder功能说明](#)
 - [17.3.7 LengthFieldPrepender功能说明](#)
- [17.4 ChannelHandler源码分析](#)
 - [17.4.1 ChannelHandler的类继承关系图](#)
 - [17.4.2 ByteToMessageDecoder源码分析](#)
 - [17.4.3 MessageToMessageDecoder源码分析](#)
 - [17.4.4 LengthFieldBasedFrameDecoder源码分析](#)
 - [17.4.5 MessageToByteEncoder源码分析](#)
 - [17.4.6 MessageToMessageEncoder源码分析](#)
 - [17.4.7 LengthFieldPrepender源码分析](#)
- [17.5 总结](#)

[第18章 EventLoop和EventLoopGroup](#)

[18.1 Netty的线程模型](#)

[18.1.1 Reactor单线程模型](#)

[18.1.2 Reactor多线程模型](#)

[18.1.3 主从Reactor多线程模型](#)

[18.1.4 Netty的线程模型](#)

[18.1.5 最佳实践](#)

[18.2 NioEventLoop源码分析](#)

[18.2.1 NioEventLoop设计原理](#)

[18.2.2 NioEventLoop继承关系类图](#)

[18.2.3 NioEventLoop](#)

[18.3 总结](#)

[第19章 Future和Promise](#)

[19.1 Future功能](#)

[19.2 ChannelFuture源码分析](#)

[19.3 Promise功能介绍](#)

[19.4 Promise源码分析](#)

[19.4.1 Promise继承关系图](#)

[19.4.2 DefaultPromise](#)

[19.5 总结](#)

[架构和行业应用篇 Netty高级特性](#)

[第20章 Java多线程编程在Netty中的应用](#)

[20.1 Java内存模型与多线程编程](#)

[20.1.1 硬件的发展和多任务处理](#)

[20.1.2 Java内存模型](#)

[20.2 Netty的并发编程实践](#)

[20.2.1 对共享的可变数据进行正确的同步](#)

[20.2.2 正确的使用锁](#)

[20.2.3 volatile的正确使用](#)

[20.2.4 CAS指令和原子类](#)

[20.2.5 线程安全类的应用](#)

[20.2.6 读写锁的应用](#)

[20.2.7 线程安全性文档说明](#)

[20.2.8 不要依赖线程优先级](#)

[20.3 总结](#)

[第21章 Netty架构剖析](#)

[21.1 Netty逻辑架构](#)

[21.1.1 Reactor通信调度层](#)

[21.1.2 职责链ChannelPipeline](#)

[21.1.3 业务逻辑编排层（Service ChannelHandler）](#)

[21.2 关键架构质量属性](#)

[21.2.1 高性能](#)

[21.2.2 可靠性](#)

[21.2.3 可定制性](#)

[21.2.4 可扩展性](#)

[21.3 总结](#)

[第22章 Netty行业应用](#)

[22.1 Netty在互联网行业的应用](#)

[22.1.1 传统垂直架构面临的问题](#)

[22.1.2 阿里分布式服务框架Dubbo](#)

[22.1.3 Dubbo的架构介绍](#)

[22.1.4 Netty在Dubbo中的应用](#)

[22.1.5 Dubbo框架集成Netty源码分析](#)

[22.2 Netty在大数据领域的应用](#)

[22.3 Netty在游戏行业的应用](#)

[22.3.1 游戏服务端架构介绍](#)

[22.3.2 Netty在游戏服务端的应用](#)

[22.4 总结](#)

[第23章 Netty未来展望](#)

[23.1 应用范围](#)

[23.2 技术演进](#)

[23.3 社区活跃度](#)

[23.4 Road Map](#)

[23.5 总结](#)

[附录 Netty参数配置表](#)

基础篇 走进Java NIO

第1章 Java的I/O演进之路

第2章 NIO入门

第1章 Java的I/O演进之路

Java是由Sun Microsystems公司在1995年首先发布的编程语言和计算平台。这项基础技术支持最新的程序，包括实用程序、游戏和业务应用程序。Java在世界各地的8.5亿多台个人计算机和数十亿套设备上运行着，其中包括移动设备和电视设备。

Java之所以能够得到如此广泛的应用，除了摆脱硬件平台的依赖具有“一次编写、到处运行”的平台无关性特性之外，另一个重要原因是：其丰富而强大的类库以及众多第三方开源类库使得基于Java语言的开发更加简单和便捷。

但是，对于一些经验丰富的程序员来说，Java的一些类库在早期设计中功能并不完善或者存在一些缺陷，其中最令人恼火的就是基于同步I/O的Socket通信类库，直到2002年2月13日JDK1.4 Merlin的发布，Java才第一次支持非阻塞I/O，这个类库的提供为JDK的通信模型带来了翻天覆地的变化。

在开始学习Netty之前，我们首先对UNIX系统常用的I/O模型进行介绍，然后对Java的I/O历史演进进行简单说明。通过本章节的学习，希望读者对同步和异步I/O以及Java的I/O类库发展有个直观的了解，方便后续章节的学习。如果你已经熟练NIO编程或者从事过UNIX网络编程，希望直接学习Java的NIO和Netty，那就可以直接跳到第2章进行学习。

本章主要包括：

- I/O基础入门

- Java的I/O演进

1.1 I/O基础入门

Java1.4之前的早期版本，Java对I/O的支持并不完善，开发人员在开发高性能I/O程序的时候，会面临一些巨大的挑战和困难，主要问题如下。

- 没有数据缓冲区，I/O性能存在问题；
- 没有C或者C++中的Channel概念，只有输入和输出流；
- 同步阻塞式I/O通信（BIO），通常会导致通信线程被长时间阻塞；
- 支持的字符集有限，硬件可移植性不好。

在Java支持异步I/O之前的很长一段时间里，高性能服务端开发领域一直被C++和C长期占据，Java的同步阻塞I/O被大家所诟病。

1.1.1 Linux网络I/O模型简介

Linux的内核将所有外部设备都看做一个文件来操作，对一个文件的读写操作会调用内核提供的系统命令，返回一个file descriptor（fd，文件描述符）。而对一个socket的读写也会有相应的描述符，称为socketfd（socket描述符），描述符就是一个数字，它指向内核中的一个结构体（文件路径，数据区等一些属性）。

根据UNIX网络编程对I/O模型的分类，UNIX提供了5种I/O模型，分别如下。

（1）阻塞I/O模型：最常用的I/O模型就是阻塞I/O模型，缺省情形下，所有文件操作都是阻塞的。我们以套接字接口为例来讲解此模型：在进程空间中调用recvfrom，其系统调用直到数据包到达且被复制到应

用进程的缓冲区中或者发生错误时才返回，在此期间一直会等待，进程在从调用recvfrom开始到它返回的整段时间内都是被阻塞的，因此被称为阻塞I/O模型，如图1-1所示。

图1-1 阻塞I/O模型

（2）非阻塞I/O模型：recvfrom从应用层到内核的时候，如果该缓冲区没有数据的话，就直接返回一个EWOULDBLOCK错误，一般都对非阻塞I/O模型进行轮询检查这个状态，看内核是不是有数据到来。如图1-2所示。

图1-2 非阻塞I/O模型

（3）I/O复用模型：Linux提供select/poll，进程通过将一个或多个fd传递给select或poll系统调用，阻塞在select操作上，这样select/poll可以帮我们侦测多个fd是否处于就绪状态。select/poll是顺序扫描fd是否就绪，而且支持的fd数量有限，因此它的使用受到了一些制约。Linux还提供了一个epoll系统调用，epoll使用基于事件驱动方式代替顺序扫描，因此性能更高。当有fd就绪时，立即回调函数rollback。如图1-3所示。

图1-3 I/O复用模型

（4）信号驱动I/O模型：首先开启套接口信号驱动I/O功能，并通过系统调用sigaction执行一个信号处理函数（此系统调用立即返回，进程继续工作，它是非阻塞的）。当数据准备就绪时，就为该进程生成一个SIGIO信号，通过信号回调通知应用程序调用recvfrom来读取数据，并通知主循环函数处理数据。如图1-4所示。

图1-4 信号驱动I/O模型

(5) 异步I/O：告知内核启动某个操作，并让内核在整个操作完成后（包括将数据从内核复制到用户自己的缓冲区）通知我们。这种模型与信号驱动模型的主要区别是：信号驱动I/O由内核通知我们何时可以开始一个I/O操作；异步I/O模型由内核通知我们I/O操作何时已经完成。如图1-5所示。

图1-5 异步I/O模型

如果想要了解更多的UNIX系统网络编程知识，可以阅读《UNIX网络编程》，里面有非常详细的原理和API介绍。对于大多数Java程序员来说，不需要了解网络编程的底层细节，大家只需要有个概念，知道对于操作系统而言，底层是支持异步I/O通信的，只不过在很长一段时间Java并没有提供异步I/O通信的类库，导致很多原生的Java程序员对这块儿比较陌生。当你了解了网络编程的基础知识后，理解Java的NIO类库就会更加容易一些。

下一个小结我们重点讲下I/O多路复用技术，因为Java NIO的核心类库多路复用器Selector就是基于epoll的多路复用技术实现。

1.1.2 I/O多路复用技术

在I/O编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者I/O多路复用技术进行处理。I/O多路复用技术通过把多个I/O的阻塞复用到同一个select的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型比，I/O多路复用的最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程的运行，降低了系统的维护工作量，节省了系统资源，I/O多路复用的主要应用场景如下。

- 服务器需要同时处理多个处于监听状态或者多个连接状态的套接字；
- 服务器需要同时处理多种网络协议的套接字。

目前支持I/O多路复用的系统调用有select、pselect、poll、epoll，在Linux网络编程过程中，很长一段时间都使用select做轮询和网络事件通知，然而select的一些固有缺陷导致了它的应用受到了很大的限制，最终Linux不得不在新的内核版本中寻找select的替代方案，最终选择了epoll。epoll与select的原理比较类似，为了克服select的缺点，epoll作了很多重大改进，现总结如下。

1. 支持一个进程打开的**socket**描述符（**FD**）不受限制（仅受限于操作系统的最大文件句柄数）。

select最大的缺陷就是单个进程所打开的FD是有一定限制的，它由FD_SETSIZE设置，默认值是1024。对于那些需要支持上万个TCP连接的大型服务器来说显然太少了。可以选择修改这个宏然后重新编译内核，不过这会带来网络效率的下降。我们也可以通过选择多进程的方案（传统的Apache方案）解决这个问题，不过虽然在Linux上创建进程的代价比较小，但仍旧是不可忽视的，另外，进程间的数据交换非常麻烦，对于Java由于没有共享内存，需要通过Socket通信或者其他方式进行数据同步，这带来了额外的性能损耗，增加了程序复杂度，所以也不是一种完美的解决方案。值得庆幸的是，epoll并没有这个限制，它所支持的FD上限是操作系统的最大文件句柄数，这个数字远远大于1024。例如，在1GB内存的机器上大约是10万个句柄左右，具体的值可以通过cat /proc/sys/fs/file-max察看，通常情况下这个值跟系统的内存关系比较大。

2. I/O效率不会随着FD数目的增加而线性下降。

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，由于网络延时或者链路空闲，任一时刻只有少部分的socket是“活跃”的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。epoll不存在这个问题，它只会对“活跃”的socket进行操作-这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的，那么，只有“活跃”的socket才会主动的去调用callback函数，其他idle状态socket则不会。在这点上，epoll实现了一个伪AIO。针对epoll和select性能对比的benchmark测试表明：如果所有的socket都处于活跃态-例如一个高速LAN环境，epoll并不比select/poll效率高太多；相反，如果过多使用epoll_ctl，效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境，epoll的效率就远在select/poll之上了。

3. 使用mmap加速内核与用户空间的消息传递。

无论是select，poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存复制就显得非常重要，epoll是通过内核和用户空间mmap同一块内存实现。

4. epoll的API更加简单。

包括创建一个epoll描述符、添加监听事件、阻塞等待所监听的事件发生，关闭epoll描述符等。

值得说明的是，用来克服select/poll缺点的方法不只有epoll，epoll只是一种Linux的实现方案。在freeBSD下有kqueue，而dev/poll是最古老的Solaris的方案，使用难度依次递增。kqueue是freebsd的宠儿，它实际上

是一个功能相当丰富的kernel事件队列，它不仅仅是select/poll的升级，而且可以处理signal、目录结构变化、进程等多种事件，kqueue是边缘触发的。/dev/poll是Solaris的产物，是这一系列高性能API中最早出现的。Kernel提供一个特殊的设备文件/dev/poll，应用程序打开这个文件得到操作fd_set的句柄，通过写入pollfd来修改它，一个特殊的ioctl调用用来替换select，不过由于出现的年代比较早，所以/dev/poll的接口实现比较原始。

到这里，I/O的基础知识已经介绍完毕，从1.2章节开始介绍Java的I/O演进历史，从BIO到NIO是Java通信类库迈出的小小一步，但却对Java在高性能通信领域的发展起到了关键性的推动作用。随着基于NIO的各类NIO框架的发展，以及基于NIO的Web服务器的发展，Java在很多领域取代了C和C++，成为企业服务端应用开发的首选语言。

1.2 Java的I/O演进

在JDK 1.4推出Java NIO之前，基于Java的所有Socket通信都采用了同步阻塞模式（BIO），这种一请求一应答的通信模型简化了上层的应用开发，但是在性能和可靠性方面却存在着巨大的瓶颈。因此，在很长一段时间里，大型的应用服务器都采用C或者C++语言开发，因为它们可以直接使用操作系统提供的异步I/O或者AIO能力。当并发访问量增大、响应时间延迟增大之后，采用Java BIO开发的服务端软件只有通过硬件的不断扩容来满足高并发和低时延，它极大地增加了企业的成本，并且随着集群规模的不断膨胀，系统的可维护性也面临巨大的挑战，只能通过采购性能更高的硬件服务器来解决问题，这会导致恶性循环。

正是由于Java传统BIO的拙劣表现，才使得Java支持非阻塞I/O的呼声日渐高涨，最终，JDK1.4版本提供了新的NIO类库，Java终于也可以支持非阻塞I/O了。

Java的I/O发展简史

从JDK1.0到JDK1.3，Java的I/O类库都非常原始，很多UNIX网络编程中的概念或者接口在I/O类库中都没有体现，例如Pipe、Channel、Buffer和Selector等。2002年发布JDK1.4时，NIO以JSR-51的身份正式随JDK发布。它新增了个java.nio包，提供了很多进行异步I/O开发的API和类库，主要的类和接口如下。

- 进行异步I/O操作的缓冲区ByteBuffer等；
- 进行异步I/O操作的管道Pipe；
- 进行各种I/O操作（异步或者同步）的Channel，包括

ServerSocketChannel和SocketChannel;

- 多种字符集的编码能力和解码能力;
- 实现非阻塞I/O操作的多路复用器selector;
- 基于流行的Perl实现的正则表达式类库;
- 文件通道FileChannel。

新的NIO类库的提供，极大地促进了基于Java的异步非阻塞编程的发展和应用，但是，它依然有不完善的地方，特别是对文件系统的处理能力仍显不足，主要问题如下。

- 没有统一的文件属性（例如读写权限）；
- API能力比较弱，例如目录的级联创建和递归遍历，往往需要自己实现；
- 底层存储系统的一些高级API无法使用；
- 所有的文件操作都是同步阻塞调用，不支持异步文件读写操作。

2011年7月28日，JDK1.7正式发布。它的一个比较大的亮点就是将原来的NIO类库进行了升级，被称为NIO2.0。NIO2.0由JSR-203演进而来，它主要提供了如下三个方面的改进。

- 提供能够批量获取文件属性的API，这些API具有平台无关性，不与特性的文件系统相耦合，另外它还提供了标准文件系统的SPI，供各个服务提供商扩展实现；
- 提供AIO功能，支持基于文件的异步I/O操作和针对网络套接字的异步操作；
- 完成JSR-51定义的通道功能，包括对配置和多播数据报的支持等。

1.3 总结

通过本章的学习，我们了解了UNIX网络编程的5种I/O模型，学习了I/O多路复用技术的基础知识。通过对Java I/O演进历史的总结和介绍，相信大家对Java的I/O演进有了一个更加直观的认识。后面的第2章节会对阻塞I/O和非阻塞I/O进行详细讲解，同时给出代码示例。相信学完第2章之后，大家就能够对传统的阻塞I/O的弊端和非阻塞I/O的优点有更加深刻的体会。好，稍微休息片刻，我们继续畅游在NIO编程的快乐海洋中！

第2章 NIO入门

在本章中，我们会分别对JDK的BIO、NIO和JDK1.7最新提供的NIO2.0的使用进行详细说明，通过流程图和代码讲解，让大家体会到随着Java I/O类库的不断发展和改进，基于Java的网络编程会变得越来越简单，随着异步I/O功能的增强，基于Java NIO开发的网络服务器甚至不逊色于采用C++开发的网络程序。

本章主要包括：

- 传统的同步阻塞式I/O编程
- 基于NIO的非阻塞编程
- 基于NIO2.0的异步非阻塞（AIO）编程
- 为什么要使用NIO编程
- 为什么选择Netty

2.1 传统的BIO编程

网络编程的基本模型是Client/Server模型，也就是两个进程之间进行相互通信，其中服务端提供位置信息（绑定的IP地址和监听端口），客户端通过连接操作向服务端监听的地址发起连接请求，通过三次握手建立连接，如果连接建立成功，双方就可以通过网络套接字（Socket）进行通信。

在基于传统同步阻塞模型开发中，ServerSocket负责绑定IP地址，启动监听端口；Socket负责发起连接操作。连接成功之后，双方通过输入和输出流进行同步阻塞式通信。

下面，我们就以经典的时间服务器（TimeServer）为例，通过代码分析来回顾和熟悉下BIO编程。

2.1.1 BIO通信模型图

首先，我们通过图2-1所示的通信模型图来熟悉下BIO的服务端通信模型：采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的一请求一应答通信模型。

图2-1 同步阻塞I/O服务端通信模型（一客户端一线程）

该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈1：1的正比关系，由于线程是Java虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性

能将急剧下降，随着并发访问量的继续增大，系统会发生线程堆栈溢出、创建新线程失败等问题，并最终导致进程宕机或者僵死，不能对外提供服务。

下面的两个小节，我们会分别对服务端和客户端进行源码分析，寻找同步阻塞I/O的弊端。

2.1.2 同步阻塞式I/O创建的TimeServer源码分析

代码清单2-1 同步阻塞I/O的TimeServer

（备注： 以下代码行号均对应源代码中实际行号。）

```
1.    package com.phei.netty.bio;
2.    import java.io.IOException;
3.    import java.net.ServerSocket;
4.    import java.net.Socket;
5.    /**
6.     * @author lilinfeng
7.     * @date 2014年2月14日
8.     * @version 1.0
9.     */
10.   public class TimeServer {
11.
12.       /**
13.        * @param args
14.        * @throws IOException
15.        */
```

```
16.         public static void main(String[] args) throws IOExc
17.         int port = 8080;
18.         if (args != null && args.length > 0) {
19.
20.             try {
21.                 port = Integer.valueOf(args[0]);
22.             } catch (NumberFormatException e) {
23.                 // 采用默认值
24.             }
25.
26.         }
27.         ServerSocket server = null;
28.         try {
29.             server = new ServerSocket(port);
30.             System.out.println("The time server is start in
31.             Socket socket = null;
32.             while (true) {
33.                 socket = server.accept();
34.                 new Thread(new TimeServerHandler(socket)).start
35.
36.             } finally {
37.                 if (server != null) {
38.                     System.out.println("The time server close");
39.                     server.close();
```

```
40.         server = null;
41.     }
42. }
43. }
44. }
```

TimeServer根据传入的参数设置监听端口，如果没有入参，使用默认值8080，29行通过构造函数创建ServerSocket，如果端口合法且没有被占用，服务端监听成功。32~35行通过一个无限循环来监听客户端的连接，如果没有客户端接入，则主线程阻塞在ServerSocket的accept操作上。启动TimeServer，通过JvisualVM打印线程堆栈，我们可以发现主程序确实阻塞在accept操作上，如图2-2所示。

图2-2 主程序线程堆栈

当有新的客户端接入的时候，执行代码34行，以Socket为参数构造TimeServerHandler对象，TimeServerHandler是一个Runnable，使用它为构造函数的参数创建一个新的客户端线程处理这条Socket链路。下面我们继续分析TimeServerHandler的代码。

代码清单2-2 同步阻塞I/O的TimeServerHandler

```
13. public class TimeServerHandler implements Runnable {
14.
15.     private Socket socket;
16.
17.     public TimeServerHandler(Socket socket) {
```

```
18.         this.socket = socket;
19.     }
20.
21.     /*
22.      * (non-Javadoc)
23.      *
24.      * @see java.lang.Runnable#run()
25.      */
26.     @Override
27.     public void run() {
28.         BufferedReader in = null;
29.         PrintWriter out = null;
30.         try {
31.             in = new BufferedReader(new InputStreamReader(
32.                 this.socket.getInputStream()));
33.             out = new PrintWriter(this.socket.getOutputStream());
34.             String currentTime = null;
35.             String body = null;
36.             while (true) {
37.                 body = in.readLine();
38.                 if (body == null)
39.                     break;
40.                 System.out.println("The time server receive ord
41.                 currentTime = "QUERY TIME ORDER".equalsIgnoreCase
42.                     System.currentTimeMillis()).toString() : "B
43.                 out.println(currentTime);
44.             }
```

```
45.
46.     } catch (Exception e) {
47.         if (in != null) {
48.             try {
49.                 in.close();
50.             } catch (IOException e1) {
51.                 e1.printStackTrace();
52.             }
53.         }
54.         if (out != null) {
55.             out.close();
56.             out = null;
57.         }
58.         if (this.socket != null) {
59.             try {
60.                 this.socket.close();
61.             } catch (IOException e1) {
62.                 e1.printStackTrace();
63.             }
64.             this.socket = null;
65.         }
66.     }
67. }
68. }
```

37行通过BufferedReader读取一行，如果已经读到了输入流的尾

部，则返回值为`null`，退出循环。如果读到了非空值，则对内容进行判断，如果请求消息为查询时间的指令"`QUERY TIME ORDER`"则获取当前最新的系统时间，通过`PrintWriter`的`println`函数发送给客户端，最后退出循环。代码47~64行释放输入流、输出流、和`Socket`套接字句柄资源，最后线程自动销毁并被虚拟机回收。

在下一个小结，我们将介绍同步阻塞I/O的客户端代码，然后分别运行服务端和客户端，查看下程序的运行结果。

2.1.3 同步阻塞式I/O创建的`TimeClient`源码分析

客户端通过`Socket`创建，发送查询时间服务器的"`QUERY TIME ORDER`"指令，然后读取服务端的响应并将结果打印出来，随后关闭连接，释放资源，程序退出执行。

代码清单2-3 同步阻塞I/O的`TimeClient`

```
13. public class TimeClient {
14.
15.     /**
16.      * @param args
17.      */
18.     public static void main(String[] args) {
19.         int port = 8080;
20.         if (args != null && args.length > 0) {
21.             try {
22.                 port = Integer.valueOf(args[0]);
23.             } catch (NumberFormatException e) {
```

```
24.         // 采用默认值
25.     }
26. }
27. Socket socket = null;
28. BufferedReader in = null;
29. PrintWriter out = null;
30. try {
31.     socket = new Socket("127.0.0.1", port);
32.     in = new BufferedReader(new InputStreamReader(
33.         socket.getInputStream()));
34.     out = new PrintWriter(socket.getOutputStream(),
35.         out.println("QUERY TIME ORDER"));
36.     System.out.println("Send order 2 server succeed
37.         String resp = in.readLine();
38.         System.out.println("Now is : " + resp);
39. } catch (Exception e) {
40.     //不需要处理
41. } finally {
42.     if (out != null) {
43.         out.close();
44.         out = null;
45.     }
46.
47.     if (in != null) {
48.         try {
49.             in.close();
50.         } catch (IOException e) {
```

```
51.             e.printStackTrace();
52.         }
53.         in = null;
54.     }
55.     if (socket != null) {
56.         try {
57.             socket.close();
58.         } catch (IOException e) {
59.             e.printStackTrace();
60.         }
61.         socket = null;
62.     }
63. }
64. }
65. }
```

第35行客户端通过PrintWriter向服务端发送"QUERY TIME ORDER"指令，然后通过BufferedReader的readLine读取响应并打印。

分别执行服务端和客户端，执行结果如下。

服务端执行结果如图2-3所示。

图2-3 同步阻塞I/O时间服务器服务端运行结果

客户端执行结果如图2-4所示。

图2-4 同步阻塞I/O时间服务器客户端运行结果

到此为止，同步阻塞式I/O开发的时间服务器程序已经讲解完毕，我们发现，BIO主要的问题在于每当有一个新的客户端请求接入时，服务端必须创建一个新的线程处理新接入的客户端链路，一个线程只能处理一个客户端连接。在高性能服务器应用领域，往往需要面向成千上万个客户端的并发连接，这种模型显然无法满足高性能、高并发接入的场景。

为了改进一线程一连接模型，后来又演进出了一种通过线程池或者消息队列实现1个或者多个线程处理N个客户端的模型，由于它的底层通信机制依然使用同步阻塞I/O，所以被称为“伪异步”，下面章节我们就对伪异步代码进行分析，看看伪异步是否能够满足我们对高性能、高并发接入的诉求。

2.2 伪异步I/O编程

为了解决同步阻塞I/O面临的一个链路需要一个线程处理的问题，后来有人对它的线程模型进行了优化，后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数 M ：线程池最大线程数 N 的比例关系，其中 M 可以远远大于 N ，通过线程池可以灵活的调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。

下面，我们结合连接模型图和源码，对伪异步I/O进行分析，看它是否能够解决同步阻塞I/O面临的问题。

2.2.1 伪异步I/O模型图

采用线程池和任务队列可以实现一种叫做伪异步的I/O通信框架，它的模型图如图2-5所示。

图2-5 伪异步I/O服务端通信模型 ($M:N$)

当有新的客户端接入的时候，将客户端的Socket封装成一个Task（该任务实现java.lang.Runnable接口）投递到后端的线程池中进行处理，JDK的线程池维护一个消息队列和 N 个活跃线程对消息队列中的任务进行处理。由于线程池可以设置消息队列的大小和最大线程数，因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。

下面的小节，我们依然采用时间服务器程序，将其改造成伪异步I/O时间服务器，然后通过对代码进行分析，找出其弊端。

2.2.2 伪异步式I/O创建的TimeServer源码分析

我们对服务端代码进行一些改造，代码如下。

代码清单2-4 伪异步I/O的TimeServer

```
13.  public class TimeServer {
14.
15.      /**
16.       * @param args
17.       * @throws IOException
18.       */
19.      public static void main(String[] args) throws IOExc
20.      int port = 8080;
21.      if (args != null && args.length > 0) {
22.          try {
23.              port = Integer.valueOf(args[0]);
24.          } catch (NumberFormatException e) {
25.              // 采用默认值
26.          }
27.      }
28.      ServerSocket server = null;
29.      try {
30.          server = new ServerSocket(port);
31.          System.out.println("The time server is start in
32.          Socket socket = null;
33.          TimeServerHandlerExecutePool singleExecutor = n
```

```
34.          50, 10000);//
```

创建I/O任务线程池

```
35.          while (true) {
36.              socket = server.accept();
37.              singleExecutor.execute(new TimeServerHandler(so

38.          }
39.      } finally {
40.          if (server != null) {
41.              System.out.println("The time server close");
42.              server.close();
43.              server = null;
44.          }
45.      }
46.  }
47. }
```

伪异步I/O的主函数代码发生了变化，我们首先创建一个时间服务器处理类的线程池，当接收到新的客户端连接的时候，将请求Socket封

装成一个Task，然后调用线程池的execute方法执行，从而避免了每个请求接入都创建一个新的线程。

代码清单2-5 伪异步I/O的TimeServerHandlerExecutePool

```
12. public class TimeServerHandlerExecutePool {
13.
14.     private ExecutorService executor;
15.
16.     public TimeServerHandlerExecutePool(int maxPoolSize
17.     executor = new ThreadPoolExecutor(Runtime.getRuntime
18.         .availableProcessors(), maxPoolSize, 120L, Time
19.         new ArrayBlockingQueue<java.lang.Runnable>(queu
20.     }
21.     public void execute(java.lang.Runnable task) {
22.         executor.execute(task);
23.     }
24. }
```

由于线程池和消息队列都是有界的，因此，无论客户端并发连接数多大，它都不会导致线程个数过于膨胀或者内存溢出，相比于传统的一连接一线程模型，是一种改良。

由于客户端代码并没有改变，因此，我们直接运行服务端和客户端，执行结果如下。

服务端运行结果如图2-6所示。

图2-6 伪异步I/O时间服务器服务端运行结果

客户端运行结果如图2-7所示。

图2-7 伪异步I/O时间服务器客户端运行结果

伪异步I/O通信框架采用了线程池实现，因此避免了为每个请求都创建一个独立线程造成的线程资源耗尽问题。但是由于它底层的通信依然采用同步阻塞模型，因此无法从根本上解决问题。下个小节我们对伪异步I/O进行深入分析，找到它的弊端，然后看看NIO是如何从根本上解决这个问题的。

2.2.3 伪异步I/O弊端分析

要对伪异步I/O的弊端进行深入分析，首先我们看两个Java同步I/O的API说明，随后我们结合代码进行详细分析。

代码清单2-6 Java输入流InputStream

```
/**
 * Reads some number of bytes from the input stream and s
 * the buffer array <code>b</code>. The number of bytes a
 * returned as an integer.
This method blocks until input data is
 * available, end of file is detected, or an exception is
```

```

*
* <p> If the length of <code>b</code> is zero, then no b
* <code>0</code> is returned; otherwise, there is an att
* least one byte. If no byte is available because the st
* end of the file, the value <code>-1</code> is returned
* least one byte is read and stored into <code>b</code>.
*
* <p> The first byte read is stored into element <code>b
* next one into <code>b[1]</code>, and so on. The number
* at most, equal to the length of <code>b</code>. Let <i>
* number of bytes actually read; these bytes will be sto
* <code>b[0]</code> through <code>b[</code><i>k</i><code>]
* leaving elements <code>b[</code><i>k</i><code>]</code>
* <code>b[b.length-1]</code> unaffected.
*
* @param

```

b the buffer into which the data is read.

```

* @return

```

the total number of bytes read into the buffer, or

```

*                   <code>-1</code> if there is no more data b

```

```
*          the stream has been reached.  
* @exception
```

IOException If the first byte cannot be read for any reason

```
* other than the end of the file, if the input stream ha  
* if some other I/O error occurs.  
* @exception
```

NullPointerException if `b` is `null`.

```
*/
```

```
public int
```

```
read(byte
```

```
b[]) throws
```

```
IOException {
```

```
return
```

```
read(b, 0, b.length);
```

```
}
```

请注意加粗斜体字部分的API说明，当对Socket的输入流进行读取操作的时候，它会一直阻塞下去，直到发生如下三种事件。

- 有数据可读；
- 可用数据已经读取完毕；
- 发生空指针或者I/O异常。

这意味着当对方发送请求或者应答消息比较缓慢、或者网络传输较慢时，读取输入流一方的通信线程将被长时间阻塞，如果对方要60s才能够将数据发送完成，读取一方的I/O线程也将会被同步阻塞60s，在此期间，其他接入消息只能在消息队列中排队。

下面我们接着对输出流进行分析，还是看JDK I/O类库输出流的API文档，然后结合文档说明进行故障分析。

代码清单2-7 Java输入流OutputStream

```
public void write(byte b[]) throws IOException
```

*Writes an array of bytes. This method will block until the b

Parameters:

b - the data to be written

Throws:

IOException

If an I/O error has occurred.

当调用OutputStream的write方法写输出流的时候，它将会被阻塞，直到所有要发送的字节全部写入完毕，或者发生异常。学习过TCP/IP相关知识的人都知道，当消息的接收方处理缓慢的时候，将不能及时地从TCP缓冲区读取数据，这将会导致发送方的TCP window size不断减小，直到为0，双方处于Keep-Alive状态，消息发送方将不能再向TCP缓冲区写入消息，这时如果采用的是同步阻塞I/O，write操作将会被无限期阻塞，直到TCP window size大于0或者发生I/O异常。

通过对输入和输出流的API文档进行分析，我们了解到读和写操作都是同步阻塞的，阻塞的时间取决于对方I/O线程的处理速度和网络I/O的传输速度。本质上来讲，我们无法保证生产环境的网络状况和对端的应用程序能足够快，如果我们的应用程序依赖对方的处理速度，它的可靠性就非常差。也许在实验室进行的性能测试结果令人满意，但是一旦上线运行，面对恶劣的网络环境和良莠不齐的第三方系统，问题就会如火山一样喷发。

伪异步I/O实际上仅仅只是对之前I/O线程模型的一个简单优化，它无法从根本上解决同步I/O导致的通信线程阻塞问题。下面我们就简单分析下如果通信对方返回应答时间过长，会引起的级联故障。

- (1) 服务端处理缓慢，返回应答消息耗费60s，平时只需要10ms。
- (2) 采用伪异步I/O的线程正在读取故障服务节点的响应，由于读取输入流是阻塞的，因此，它将会被同步阻塞60s。
- (3) 假如所有的可用线程都被故障服务器阻塞，那后续所有的I/O消息都将在队列中排队。
- (4) 由于线程池采用阻塞队列实现，当队列积满之后，后续入队

列的操作将被阻塞。

（5）由于前端只有一个Accptor线程接收客户端接入，它被阻塞在线程池的同步阻塞队列之后，新的客户端请求消息将被拒绝，客户端会发生大量的连接超时。

（6）由于几乎所有的连接都超时，调用者会认为系统已经崩溃，无法接收新的请求消息。

如何破解这个难题？下节的NIO将给出答案。

2.3 NIO编程

在介绍NIO编程之前，我们首先需要澄清一个概念：NIO到底是什么的简称？有人称之为New I/O，因为它相对于之前的I/O类库是新增的，所以被称为New I/O，这是它的官方叫法。但是，由于之前老的I/O类库是阻塞I/O，New I/O类库的目标就是要让Java支持非阻塞I/O，所以，更多的人喜欢称之为非阻塞I/O（Non-block I/O），由于非阻塞I/O更能够体现NIO的特点，所以本书使用的NIO都指的是非阻塞I/O。

与Socket类和ServerSocket类相对应，NIO也提供了SocketChannel和ServerSocketChannel两种不同的套接字通道实现。这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单，但是性能和可靠性都不好，非阻塞模式则正好相反。开发人员一般可以根据自己的需要来选择合适的模式，一般来说，低负载、低并发的应用程序可以选择同步阻塞I/O以降低编程复杂度，但是对于高负载、高并发的网络应用，需要使用NIO的非阻塞模式进行开发。

下面的小节首先介绍NIO编程中的一些基本概念，然后通过NIO服务端的序列图和源码讲解，让大家快速地熟悉NIO编程的关键步骤和API的使用。如果你已经熟悉了NIO编程，可以跳过2.3节直接学习后面的章节。

2.3.1 NIO类库简介

新的输入/输出（NIO）库是在JDK 1.4中引入的。NIO弥补了原来同步阻塞I/O的不足，它在标准Java代码中提供了高速的、面向块的I/O。通过定义包含数据的类，以及通过以块的形式处理这些数据，NIO不用

使用本机代码就可以利用低级优化，这是原来的I/O包所无法做到的。下面我们对NIO的一些概念和功能做下简单介绍，以便大家能够快速地了解NIO类库和相关概念。

1. 缓冲区Buffer

我们首先介绍缓冲区（Buffer）的概念，Buffer是一个对象，它包含一些要写入或者要读出的数据。在NIO类库中加入Buffer对象，体现了新库与原I/O的一个重要区别。在面向流的I/O中，可以将数据直接写入或者将数据直接读到Stream对象中。

在NIO库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。

缓冲区实质上是一个数组。通常它是一个字节数组（ByteBuffer），也可以使用其他种类的数组。但是一个缓冲区不仅仅是一个数组，缓冲区提供了对数据的结构化访问以及维护读写位置（limit）等信息。

最常用的缓冲区是ByteBuffer，一个ByteBuffer提供了一组功能用于操作byte数组。除了ByteBuffer，还有其他的一些缓冲区，事实上，每一种Java基本类型（除了Boolean类型）都对应有一种缓冲区，具体如下。

- ByteBuffer: 字节缓冲区
- CharBuffer: 字符缓冲区
- ShortBuffer: 短整型缓冲区

- **IntBuffer**: 整形缓冲区
- **LongBuffer**: 长整形缓冲区
- **FloatBuffer**: 浮点型缓冲区
- **DoubleBuffer**: 双精度浮点型缓冲区

缓冲区的类图继承关系如图2-8所示。

图2-8 Buffer继承关系图

每一个Buffer类都是Buffer接口的一个子实例。除了ByteBuffer，每一个Buffer类都有完全一样的操作，只是它们所处理的数据类型不一样。因为大多数标准I/O操作都使用ByteBuffer，所以它除了具有一般缓冲区的操作之外还提供一些特有的操作，方便网络读写。

2. 通道Channel

Channel是一个通道，可以通过它读取和写入数据，它就像自来水管一样，网络数据通过Channel读取和写入。通道与流的不同之处在于通道是双向的，流只是在一个方向上移动（一个流必须是InputStream或者OutputStream的子类），而且通道可以用于读、写或者同时用于读写。

因为Channel是全双工的，所以它可以比流更好地映射底层操作系统的API。特别是在UNIX网络编程模型中，底层操作系统的通道都是全双工的，同时支持读写操作。

Channel的类图继承关系如图2-9所示。

图2-9 Channel继承关系类图

自顶向下看，前三层主要是Channel接口，用于定义它的功能，后面是一些具体的功能类（抽象类），从类图可以看出，实际上Channel可以分为两大类：分别是用于网络读写的SelectableChannel和用于文件操作的FileChannel。

本书涉及的ServerSocketChannel和SocketChannel都是SelectableChannel的子类，关于它们的具体用法将在后续的代码中体现。

3. 多路复用器Selector

在本节中，我们将探索多路复用器Selector，它是Java NIO编程的基础，熟练地掌握Selector对于掌握NIO编程至关重要。多路复用器提供选择已经就绪的任务的能力。简单来讲，Selector会不断地轮询注册在其上的Channel，如果某个Channel上面有新的TCP连接接入、读和写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的I/O操作。

一个多路复用器Selector可以同时轮询多个Channel，由于JDK使用了epoll()代替传统的select实现，所以它并没有最大连接句柄1024/2048的限制。这也就意味着只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端，这确实是个非常巨大的进步。

下面，我们通过NIO编程的序列图和源码分析来熟悉相关的概念，以便巩固我们前面所学的NIO基础知识。

2.3.2 NIO服务端序列图

NIO服务端通信序列图如图2-10所示。

图2-10 NIO服务端通信序列图

下面，我们对NIO服务端的主要创建过程进行讲解和说明，作为NIO的基础入门，我们将忽略掉一些在生产环境中部署所需要的一些特性和功能。

步骤一：打开ServerSocketChannel，用于监听客户端的连接，它是所有客户端连接的父管道，代码示例如下。

```
ServerSocketChannel acceptorSvr = ServerSocketChannel.open();
```

步骤二：绑定监听端口，设置连接为非阻塞模式，示例代码如下。

```
acceptorSvr.socket().bind(new InetSocketAddress(InetAddress.g  
acceptorSvr.configureBlocking(false);
```

步骤三：创建Reactor线程，创建多路复用器并启动线程，代码如下。

```
Selector selector = Selector.open();  
New Thread(new ReactorTask()).start();
```

步骤四：将ServerSocketChannel注册到Reactor线程的多路复用器Selector上，监听ACCEPT事件，代码如下。

```
SelectionKey key = acceptorSvr.register( selector, SelectionK
```

步骤五：多路复用器在线程run方法的无限循环体内轮询准备就绪的Key，代码如下。

```
int num = selector.select();
Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();
while (it.hasNext()) {
    SelectionKey key = (SelectionKey)it.next();
    // ... deal with I/O event ...
}
```

步骤六：多路复用器监听到有新的客户端接入，处理新的接入请求，完成TCP三次握手，建立物理链路，代码示例如下。

```
SocketChannel channel = svrChannel.accept();
```

步骤七：设置客户端链路为非阻塞模式，示例代码如下。

```
channel.configureBlocking(false);
channel.socket().setReuseAddress(true);
.....
```

步骤八：将新接入的客户端连接注册到Reactor线程的多路复用器上，监听读操作，用来读取客户端发送的网络消息，代码如下。

```
SelectionKey key = socketChannel.register( selector, Selectio
```

步骤九：异步读取客户端请求消息到缓冲区，示例代码如下。

```
int readNumber = channel.read(receivedBuffer);
```

步骤十：对ByteBuffer进行编解码，如果有半包消息指针reset，继续读取后续的报文，将解码成功的消息封装成Task，投递到业务线程池中，进行业务逻辑编排，示例代码如下。

```
Object message = null;
while(buffer.hasRemain())
{
    byteBuffer.mark();
    Object message = decode(byteBuffer);
    if (message == null)
    {
        byteBuffer.reset();
        break;
    }
    messageList.add(message );
}
if (!byteBuffer.hasRemain())
byteBuffer.clear();
else
```

```
        byteBuffer.compact();
    if (messageList != null & !messageList.isEmpty())
    {
        for(Object messageE : messageList)
            handlerTask(messageE);
    }
```

步骤十一：将POJO对象encode成ByteBuffer，调用SocketChannel的异步write接口，将消息异步发送给客户端，示例代码如下。

```
socketChannel.write(buffer);
```

注意：如果发送区TCP缓冲区满，会导致写半包，此时，需要注册监听写操作位，循环写，直到整包消息写入TCP缓冲区，此处不赘述，后续Netty源码分析章节会详细分析Netty的处理策略。

当我们了解创建NIO服务端的基本步骤之后，下面我们将前面的时间服务器程序通过NIO重写一遍，让大家能够学习到完整版的NIO服务端创建。

2.3.3 NIO创建的TimeServer源码分析

我们将在TimeServer例程中给出完整的NIO创建的时间服务器源码。

代码清单2-8 NIO时间服务器TimeServer

```

9.    public class TimeServer {
10.
11.        /**
12.         * @param args
13.         * @throws IOException
14.         */
15.        public static void main(String[] args) throws IOExc
16.        int port = 8080;
17.        if (args != null && args.length > 0) {
18.            try {
19.                port = Integer.valueOf(args[0]);
20.            } catch (NumberFormatException e) {
21.                // 采用默认值
22.            }
23.        }
24.        MultiplexerTimeServer timeServer = new MultiplexerT
25.        New Thread(timeServer, "NIO-MultiplexerTimeServer-0
26.        }
27.    }

```

我们对NIO创建的TimeServer进行简单分析下，16~23行跟之前的一样，设置监听端口。24~25行创建了一个被称为MultiplexerTimeServer的多路复用类，它是个一个独立的线程，负责轮询多路复用器Selector，可以处理多个客户端的并发接入。现在我们继续看MultiplexerTimeServer的源码。

代码清单2-8 NIO时间服务器MultiplexerTimeServer

```
17. public class MultiplexerTimeServer implements Runnable
18.
19.     private Selector selector;
20.
21.     private ServerSocketChannel servChannel;
22.
23.     private volatile boolean stop;
24.
25.     /**
26.      * 初始化多路复用器、绑定监听端口
27.      *
28.      * @param port
29.      */
30.     public MultiplexerTimeServer(int port) {
31.     try {
32.         selector = Selector.open();
33.         servChannel = ServerSocketChannel.open();
34.         servChannel.configureBlocking(false);
35.         servChannel.socket().bind(new InetSocketAddress
36.         servChannel.register(selector, SelectionKey.OP_
37.         System.out.println("The time server is start in
38.     } catch (IOException e) {
39.         e.printStackTrace();
40.         System.exit(1);
41.     }
42.     }
```



```
70.             key.channel().close();
71.         }
72.     }
73. }
74. } catch (Throwable t) {
75.     t.printStackTrace();
76. }
77. }
78.
79. // 多路复用器关闭后，所有注册在上面的Channel和Pipe等资源都
80. if (selector != null)
81.     try {
82.         selector.close();
83.     } catch (IOException e) {
84.         e.printStackTrace();
85.     }
86. }
87.
88. private void handleInput(SelectionKey key) throws I
89.
90. if (key.isValid()) {
91.     // 处理新接入的请求消息
92.     if (key.isAcceptable()) {
93.         // Accept the new connection
94.         ServerSocketChannel ssc = (ServerSocketChannel)
95.         SocketChannel sc = ssc.accept();
96.         sc.configureBlocking(false);
```

```
97.         // Add the new connection to the selector
98.         sc.register(selector, SelectionKey.OP_READ);
99.     }
100.        if (key.isReadable()) {
101.            // Read the data
102.            SocketChannel sc = (SocketChannel) key.channel();
103.            ByteBuffer readBuffer = ByteBuffer.allocate(1024);
104.            int readBytes = sc.read(readBuffer);
105.            if (readBytes > 0) {
106.                readBuffer.flip();
107.                byte[] bytes = new byte[readBuffer.remaining()];
108.                readBuffer.get(bytes);
109.                String body = new String(bytes, "UTF-8");
110.                System.out.println("The time server received: "
111.                    + body);
112.                String currentTime = "QUERY TIME ORDER"
113.                    .equalsIgnoreCase(body) ? new java.util.Date(
114.                        System.currentTimeMillis()).toString()
115.                    : "BAD ORDER";
116.                doWrite(sc, currentTime);
117.            } else if (readBytes < 0) {
118.                // 对端链路关闭
119.                key.cancel();
120.                sc.close();
121.            } else
122.                ; // 读到0字节, 忽略
123.        }
```

```
124.      }
125.      }
126.
127.      private void doWrite(SocketChannel channel, String
128.          throws IOException {
129.      if (response != null && response.trim().length()
130.          byte[] bytes = response.getBytes();
131.          ByteBuffer writeBuffer = ByteBuffer.allocate(
132.          writeBuffer.put(bytes);
133.          writeBuffer.flip();
134.          channel.write(writeBuffer);
135.      }
136.      }
137.  }
```

由于这个类相比于传统的Socket编程会稍微复杂一些，在此展开进行详细分析，我们从如下几个关键步骤来讲解多路复用处理类。

(1) 30~42行为构造方法，在构造方法中进行资源初始化，创建多路复用器Selector、ServerSocketChannel，对Channel和TCP参数进行配置。例如，将ServerSocketChannel设置为异步非阻塞模式，它的backlog设置为1024。系统资源初始化成功后，将ServerSocket Channel注册到Selector，监听SelectionKey.OP_ACCEPT操作位；如果资源初始化失败（例如端口被占用），则退出。

(2) 55~77行在线程的run方法的while循环体中循环遍历selector，它的休眠时间为1s，无论是否有读写等事件发生，selector每隔1s都被唤

醒一次，`selector`也提供了一个无参的`select`方法。当有处于就绪状态的`Channel`时，`selector`将返回就绪状态的`Channel`的`SelectionKey`集合，通过对就绪状态的`Channel`集合进行迭代，可以进行网络的异步读写操作。

（3）92~99行处理新接入的客户端请求消息，根据`SelectionKey`的操作位进行判断即可获知网络事件的类型，通过`ServerSocketChannel`的`accept`接收客户端的连接请求并创建`SocketChannel`实例，完成上述操作后，相当于完成了TCP的三次握手，TCP物理链路正式建立。注意，我们需要将新创建的`SocketChannel`设置为异步非阻塞，同时也可以对其TCP参数进行设置，例如TCP接收和发送缓冲区的大小等，作为入门的例子，例程没有进行额外的参数设置。

（4）100~125行用于读取客户端的请求消息，首先创建一个`ByteBuffer`，由于我们事先无法得知客户端发送的码流大小，作为例程，我们开辟一个1M的缓冲区。然后调用`SocketChannel`的`read`方法读取请求码流。注意，由于我们已经将`SocketChannel`设置为异步非阻塞模式，因此它的`read`是非阻塞的。使用返回值进行判断，看读取到的字节数，返回值有以下三种可能的结果。

- 返回值大于0：读到了字节，对字节进行编解码；
- 返回值等于0：没有读取到字节，属于正常场景，忽略；
- 返回值为-1：链路已经关闭，需要关闭`SocketChannel`，释放资源。

当读取到码流以后，我们进行解码，首先对`readBuffer`进行`flip`操作，它的作用是将缓冲区当前的`limit`设置为`position`，`position`设置为0，用于后续对缓冲区的读取操作。然后根据缓冲区可读的字节个数创建字节数组，调用`ByteBuffer`的`get`操作将缓冲区可读的字节数组复制到新创建的字节数组中，最后调用字符串的构造函数创建请求消息体并打印。

如果请求指令是"QUERY TIME ORDER"则把服务器的当前时间编码后返回给客户端，下面我们看看异步发送应答消息给客户端的情况。

（5）127~135行将应答消息异步发送给客户端。我们看下关键代码，首先将字符串编码成字节数组，根据字节数组的容量创建ByteBuffer，调用ByteBuffer的put操作将字节数组复制到缓冲区中，然后对缓冲区进行flip操作，最后调用SocketChannel的write方法将缓冲区中的字节数组发送出去。需要指出的是，由于SocketChannel是异步非阻塞的，它并不保证一次能够把需要发送的字节数组发送完，此时会出现“写半包”问题，我们需要注册写操作，不断轮询Selector将没有发送完的ByteBuffer发送完毕，可以通过ByteBuffer的hasRemain()方法判断消息是否发送完成。此处仅仅是个简单的入门级例程，没有演示如何处理“写半包”场景，后续的章节会有详细说明。

使用NIO创建TimeServer服务器完成之后，我们继续学习如何创建NIO客户端。首先还是通过时序图了解关键步骤和过程，然后结合代码进行详细分析。

2.3.4 NIO客户端序列图

NIO客户端创建序列图如图2-11所示。

图2-11 NIO客户端创建序列图

步骤一：打开SocketChannel，绑定客户端本地地址（可选，默认系统会随机分配一个可用的本地地址），示例代码如下。

```
SocketChannel clientChannel = SocketChannel.open();
```

步骤二：设置SocketChannel为非阻塞模式，同时设置客户端连接的TCP参数，示例代码如下。

```
clientChannel.configureBlocking(false);
socket.setReuseAddress(true);
socket.setReceiveBufferSize(BUFFER_SIZE);
socket.setSendBufferSize(BUFFER_SIZE);
```

步骤三：异步连接服务端，示例代码如下。

```
boolean connected=clientChannel.connect(new InetSocketAddress
```

步骤四：判断是否连接成功，如果连接成功，则直接注册读状态位到多路复用器中，如果当前没有连接成功（异步连接，返回false，说明客户端已经发送sync包，服务端没有返回ack包，物理链路还没有建立），示例代码如下。

```
if (connected)
{
    clientChannel.register( selector, SelectionKey.OP_READ, i
}
else
{
    clientChannel.register( selector, SelectionKey.OP_CONNECT
}
```

步骤五：向Reactor线程的多路复用器注册OP_CONNECT状态位，监听服务端的TCP ACK应答，示例代码如下。

```
clientChannel.register( selector, SelectionKey.OP_CONNECT, io
```

步骤六：创建Reactor线程，创建多路复用器并启动线程，代码如下。

```
Selector selector = Selector.open();  
New Thread(new ReactorTask()).start();
```

步骤七：多路复用器在线程run方法的无限循环体内轮询准备就绪的Key，代码如下。

```
int num = selector.select();  
Set selectedKeys = selector.selectedKeys();  
Iterator it = selectedKeys.iterator();  
while (it.hasNext()) {  
    SelectionKey key = (SelectionKey)it.next();  
    // ... deal with I/O event ...  
}
```

步骤八：接收connect事件进行处理，示例代码如下。

```
if (key.isConnectable())
```

```
//handlerConnect();
```

步骤九：判断连接结果，如果连接成功，注册读事件到多路复用器，示例代码如下。

```
if (channel.finishConnect())  
    registerRead();
```

步骤十：注册读事件到多路复用器，示例代码如下。

```
clientChannel.register( selector, SelectionKey.OP_READ, ioHan
```

步骤十一：异步读客户端请求消息到缓冲区，示例代码如下。

```
int  readNumber =  channel.read(receivedBuffer);
```

步骤十二：对ByteBuffer进行编解码，如果有半包消息接收缓冲区Reset，继续读取后续的报文，将解码成功的消息封装成Task，投递到业务线程池中，进行业务逻辑编排，示例代码如下。

```
Object message = null;  
while(buffer.hasRemain())  
{  
    byteBuffer.mark();  
    Object message = decode(byteBuffer);
```

```
        if (message == null)
        {
            byteBuffer.reset();
            break;
        }
        messageList.add(message );
    }
    if (!byteBuffer.hasRemain())
    byteBuffer.clear();
    else
        byteBuffer.compact();
    if (messageList != null & !messageList.isEmpty())
    {
        for(Object messageE : messageList)
            handlerTask(messageE);
    }
}
```

步骤十三：将POJO对象encode成ByteBuffer，调用SocketChannel的异步write接口，将消息异步发送给客户端，示例代码如下。

```
socketChannel.write(buffer);
```

通过序列图和关键代码的解说，相信大家对创建NIO客户端程序已经有了一个初步的了解，下面就跟随着我们的脚步，继续看看如何使用NIO改造之前的时间服务器客户端TimeClient吧。

2.3.5 NIO创建的TimeClient源码分析

我们首先还是看下如何对TimeClient进行改造。

代码清单2-9 NIO时间服务器客户端TimeClient

```
16.         if (args != null && args.length > 0) {
17.             try {
18.                 port = Integer.valueOf(args[0]);
19.             } catch (NumberFormatException e) {
20.                 // 采用默认值
21.             }
22.         }
23.         new Thread(new TimeClientHandle("127.0.0.1", port),
24.             .start());
25.     }
26. }
```

与之前唯一不同的地方在于通过创建TimeClientHandle线程来处理异步连接和读写操作，由于TimeClient非常简单且变更不大，这里重点分析TimeClientHandle，代码如下。

代码清单2-10 NIO时间服务器客户端TimeClientHandle

```
1.     package com.phei.netty.nio;
2.     import java.io.IOException;
3.     import java.net.InetSocketAddress;
```

```
4.    import java.nio.ByteBuffer;
5.    import java.nio.channels.SelectionKey;
6.    import java.nio.channels.Selector;
7.    import java.nio.channels.SocketChannel;
8.    import java.util.Iterator;
9.    import java.util.Set;
10.
11.    /**
12.     * @author Administrator
13.     * @date 2014年2月16日
14.     * @version 1.0
15.     */
16.    public class TimeClientHandle implements Runnable {
17.        private String host;
18.        private int port;
19.        private Selector selector;
20.        private SocketChannel socketChannel;
21.        private volatile boolean stop;
22.
23.        public TimeClientHandle(String host, int port) {
24.            this.host = host == null ? "127.0.0.1" : host;
25.            this.port = port;
26.            try {
27.                selector = Selector.open();
28.                socketChannel = SocketChannel.open();
29.                socketChannel.configureBlocking(false);
30.            } catch (IOException e) {
```

```
31.         e.printStackTrace();
32.         System.exit(1);
33.     }
34. }
35.
36.  /*
37.   * (non-Javadoc)
38.   *
39.   * @see java.lang.Runnable#run()
40.   */
41. @Override
42. public void run() {
43.     try {
44.         doConnect();
45.     } catch (IOException e) {
46.         e.printStackTrace();
47.         System.exit(1);
48.     }
49.     while (!stop) {
50.         try {
51.             selector.select(1000);
52.             Set<SelectionKey> selectedKeys = selector.select
53.             Iterator<SelectionKey> it = selectedKeys.iterator
54.             SelectionKey key = null;
55.             while (it.hasNext()) {
56.                 key = it.next();
57.                 it.remove();
```

```
58.         try {
59.             handleInput(key);
60.         } catch (Exception e) {
61.             if (key != null) {
62.                 key.cancel();
63.                 if (key.channel() != null)
64.                     key.channel().close();
65.             }
66.         }
67.     }
68. } catch (Exception e) {
69.     e.printStackTrace();
70.     System.exit(1);
71. }
72. }
73.
74. // 多路复用器关闭后，所有注册在上面的Channel和Pipe等资源都
75. if (selector != null)
76.     try {
77.         selector.close();
78.     } catch (IOException e) {
79.         e.printStackTrace();
80.     }
81. }
82.
83. private void handleInput(SelectionKey key) throws I
84.
```

```
85.         if (key.isValid()) {
86.             // 判断是否连接成功
87.             SocketChannel sc = (SocketChannel) key.channel()
88.             if (key.isConnectable()) {
89.                 if (sc.finishConnect()) {
90.                     sc.register(selector, SelectionKey.OP_READ)
91.                     doWrite(sc);
92.                 } else
93.                     System.exit(1); // 连接失败，进程退出
94.             }
95.             if (key.isReadable()) {
96.                 ByteBuffer readBuffer = ByteBuffer.allocate(102
97.                 int readBytes = sc.read(readBuffer);
98.                 if (readBytes > 0) {
99.                     readBuffer.flip();
100.                     byte[] bytes = new byte[readBuffer.remaining()];
101.                     readBuffer.get(bytes);
102.                     String body = new String(bytes, "UTF-8");
103.                     System.out.println("Now is : " + body);
104.                     this.stop = true;
105.                 } else if (readBytes < 0) {
106.                     // 对端链路关闭
107.                     key.cancel();
108.                     sc.close();
109.                 } else
110.                     ; // 读到0字节，忽略
111.             }
```

```

112.     }
113.
114.     }
115.
116.     private void doConnect() throws IOException {
117.         // 如果直接连接成功，则注册到多路复用器上，发送请求消息，讨
118.         if(socketChannel.connect(new InetSocketAddress(ho
119.             socketChannel.register(selector, SelectionKey
120.             doWrite(socketChannel);
121.     } else
122.         socketChannel.register(selector, SelectionKey
123.     }
124.
125.     private void doWrite(SocketChannel sc) throws IOE
126.     byte[] req = "QUERY TIME ORDER".getBytes();
127.     ByteBuffer writeBuffer = ByteBuffer.allocate(req.
128.     writeBuffer.put(req);
129.     writeBuffer.flip();
130.     sc.write(writeBuffer);
131.     if (!writeBuffer.hasRemaining())
132.         System.out.println("Send order 2 server succe
133.     }
134. }

```

与服务端类似，接下来我们通过对关键步骤的源码进行分析和解读，让大家深入了解如何创建NIO客户端以及如何使用NIO的API。

(1) 23~34行构造函数用于初始化NIO的多路复用器和SocketChannel对象。需要注意的是，创建SocketChannel之后，需要将其设置为异步非阻塞模式。就像在2.3.3章节中所讲的，我们可以设置SocketChannel的TCP参数，例如接收和发送的TCP缓冲区大小。

(2) 43~48行用于发送连接请求，作为示例，连接是成功的，所以不需要做重连操作，因此将其放到循环之前。下面我们具体看看doConnect的实现，代码跳到第116~123行，首先对SocketChannel的connect()操作进行判断，如果连接成功，则将SocketChannel注册到多路复用器Selector上，注册SelectionKey.OP_READ，如果没有直接连接成功，则说明服务端没有返回TCP握手应答消息，但这并不代表连接失败，我们需要将SocketChannel注册到多路复用器Selector上，注册SelectionKey.OP_CONNECT，当服务端返回TCP syn-ack消息后，Selector就能够轮询到这个SocketChannel处于连接就绪状态。

(3) 49~72行在循环体中轮询多路复用器Selector，当有就绪的Channel时，执行第59行的handleInput(key)方法，下面我们就对handleInput方法进行分析。

(4) 跳到第83行，我们首先对SelectionKey进行判断，看它处于什么状态。如果是处于连接状态，说明服务端已经返回ACK应答消息。这时我们需要对连接结果进行判断，调用SocketChannel的finishConnect()方法，如果返回值为true，说明客户端连接成功；如果返回值为false或者直接抛出IOException，说明连接失败。在本例程中，返回值为true，说明连接成功。将SocketChannel注册到多路复用器上，注册SelectionKey.OP_READ操作位，监听网络读操作，然后发送请求消息给服务端。

下面我们对doWrite(sc)进行分析。代码跳到125行，我们构造请求消息体，然后对其编码，写入到发送缓冲区中，最后调用SocketChannel的write方法进行发送。由于发送是异步的，所以会存在“半包写”问题，此处不再赘述。最后通过hasRemaining()方法对发送结果进行判断，如果缓冲区中的消息全部发送完成，打印"Send order 2 server succeed."

(5) 返回代码第95行，我们继续分析客户端是如何读取时间服务器应答消息的。如果客户端接收到了服务端的应答消息，则SocketChannel是可读的，由于无法事先判断应答码流的大小，我们就预分配1M的接收缓冲区用于读取应答消息，调用SocketChannel的read()方法进行异步读取操作。由于是异步操作，所以必须对读取的结果进行判断，这部分的处理逻辑已经在2.3.3章节详细介绍过，此处不再赘述。如果读取到了消息，则对消息进行解码，最后打印结果。执行完成后将stop置为true，线程退出循环。

(6) 线程退出循环后，我们需要对连接资源进行释放，以实现“优雅退出”。75~80行用于多路复用器的资源释放，由于多路复用器上可能注册成千上万的Channel或者pipe，如果一一对这些资源进行释放显然不合适。因此，JDK底层会自动释放所有跟此多路复用器关联的资源，JDK的API DOC如图2-12所示。

图2-12 多路复用器Selector的资源释放

到此为止，我们已经通过NIO对时间服务器完成了改造，并对源码进行了分析和解读，下面分别执行时间服务器的服务端和客户端，看执行结果。

服务端执行结果如图2-13所示。

图2-13 NIO时间服务器服务端执行结果

客户端执行结果如图2-14所示。

图2-14 NIO时间服务器客户端执行结果

通过源码对比分析，我们发现NIO编程难度确实比同步阻塞BIO大很多，我们的NIO例程并没有考虑“半包读”和“半包写”，如果加上这些，代码将会更加复杂。NIO代码既然这么复杂，为什么它的应用却越来越广泛呢，使用NIO编程的优点总结如下。

（1）客户端发起的连接操作是异步的，可以通过在多路复用器注册OP_CONNECT等待后续结果，不需要像之前的客户端那样被同步阻塞。

（2）SocketChannel的读写操作都是异步的，如果没有可读写的数
据它不会同步等待，直接返回，这样I/O通信线程就可以处理其他的链路，不需要同步等待这个链路可用。

（3）线程模型的优化：由于JDK的Selector在Linux等主流操作系统上通过epoll实现，它没有连接句柄数的限制（只受限于操作系统的最大句柄数或者对单个进程的句柄限制），这意味着一个Selector线程可以同时处理成千上万个客户端连接，而且性能不会随着客户端的增加而线性下降，因此，它非常适合做高性能、高负载的网络服务器。

JDK1.7升级了NIO类库，升级后的NIO类库被称为NIO2.0，引人注目的是，Java正式提供了异步文件I/O操作，同时提供了与UNIX网络编程事件驱动I/O对应的AIO，下面的2.4章节我们学习下如何利用NIO2.0编写AIO程序，还是以时间服务器为例进行讲解。

2.4 AIO编程

NIO2.0引入了新的异步通道的概念，并提供了异步文件通道和异步套接字通道的实现。异步通道提供两种方式获取操作结果。

- 通过`java.util.concurrent.Future`类来表示异步操作的结果；
- 在执行异步操作的时候传入一个`java.nio.channels`。

`CompletionHandler`接口的实现类作为操作完成的回调。

NIO2.0的异步套接字通道是真正的异步非阻塞I/O，它对应UNIX网络编程中的事件驱动I/O（AIO），它不需要通过多路复用器（`Selector`）对注册的通道进行轮询操作即可实现异步读写，从而简化了NIO的编程模型。

下面通过代码来熟悉NIO2.0 AIO的相关类库，仍旧以时间服务器为例程进行讲解。

2.4.1 AIO创建的TimeServer源码分析

首先看下时间服务器的主函数。

代码清单2-11 AIO时间服务器服务端TimeClientHandle

```
10. public class TimeServer {
11.
12.     /**
13.      * @param args
```

```
14.      * @throws IOException
15.      */
16.      public static void main(String[] args) throws IOExc
17.      int port = 8080;
18.      if (args != null && args.length > 0) {
19.          try {
20.              port = Integer.valueOf(args[0]);
21.          } catch (NumberFormatException e) {
22.              // 采用默认值
23.          }
24.      }
25.      AsyncTimeServerHandler timeServer=new AsyncTimeServ
26.      new Thread(timeServer, "AIO-AsyncTimeServerHandler-
27.      }
28.  }
```

我们直接从第25行开始看，首先创建异步的时间服务器处理类，然后启动线程将AsyncTimeServerHandler拉起，代码如下。

代码清单2-12 AIO时间服务器服务端

```
13.  public class AsyncTimeServerHandler implements Runnable
14.
15.      private int port;
16.
17.      CountdownLatch latch;
```

```
18.      AsynchronousServerSocketChannel asynchronousServers
19.
20.      public AsyncTimeServerHandler(int port) {
21.          this.port = port;
22.          try {
23.              asynchronousServerSocketChannel = AsynchronousS
24.                  .open();
25.              asynchronousServerSocketChannel.bind(new InetSo
26.                  System.out.println("The time server is start in
27.          } catch (IOException e) {
28.              e.printStackTrace();
29.          }
30.      }
31.
32.      /*
33.       * (non-Javadoc)
34.       *
35.       * @see java.lang.Runnable#run()
36.       */
37.      @Override
38.      public void run() {
39.
40.          latch = new CountDownLatch(1);
41.          doAccept();
42.          try {
43.              latch.await();
44.          } catch (InterruptedException e) {
```

```
45.          e.printStackTrace();
46.      }
47.  }
48.
49.      public void doAccept() {
50.          asynchronousServerSocketChannel.accept(this,
51.              new AcceptCompletionHandler());
52.      }
```

我们重点对AsyncTimeServerHandler进行分析。首先看20~27行，在构造方法中，我们首先创建一个异步的服务端通道AsynchronousServerSocketChannel，然后调用它的bind方法绑定监听端口，如果端口合法且没被占用，绑定成功，打印启动成功提示到控制台。

在线程的run方法中，第40行我们初始化CountDownLatch对象，它的作用是在完成一组正在执行的操作之前，允许当前的线程一直阻塞。在本例程中，我们让线程在此阻塞，防止服务端执行完成退出。在实际项目应用中，不需要启动独立的线程来处理AsynchronousServerSocketChannel，这里仅仅是个demo演示。

第41行用于接收客户端的连接，由于是异步操作，我们可以传递一个CompletionHandler <AsynchronousSocketChannel,? super A>类型的handler实例接收accept操作成功的通知消息，在本例程中我们通过AcceptCompletionHandler实例作为handler来接收通知消息，下面继续对AcceptCompletionHandler进行分析。

代码清单2-13 AIO时间服务器服务端AcceptCompletionHandler

```
14.  
15.      @Override  
16.      public void completed(AsynchronousSocketChannel res  
17.          AsyncTimeServerHandler attachment) {  
18.          attachment.asynchronousServerSocketChannel.accept(a  
19.              ByteBuffer buffer = ByteBuffer.allocate(1024);  
20.              result.read(buffer, buffer, new ReadCompletionHandl  
21.          }  
22.  
23.      @Override  
24.      public void failed(Throwable exc, AsyncTimeServerHan  
25.          exc.printStackTrace();  
26.          attachment.latch.countDown();  
27.      }  
28.  }
```

CompletionHandler有两个方法，分别如下。

- `public void completed(AsynchronousSocketChannel result, AsyncTimeServerHandler attachment);`
- `public void failed(Throwable exc, AsyncTimeServerHandler attachment)。`

下面我们分别对这两个接口的实现进行分析。首先看completed接口的实现，代码18~20行，我们从attachment获取成员变量

AsynchronousServerSocketChannel，然后继续调用它的accept方法。有的读者在此可能会心存疑惑：既然已经接收客户端成功了，为什么还要再次调用accept方法呢？原因是这样的：当我们调用AsynchronousServerSocketChannel的accept方法后，如果有新的客户端连接接入，系统将回调我们传入的CompletionHandler实例的completed方法，表示新的客户端已经接入成功，因为一个AsynchronousServerSocketChannel可以接收成千上万个客户端，所以我们需要继续调用它的accept方法，接收其他的客户端连接，最终形成一个循环。每当接收一个客户端连接成功之后，再异步接收新的客户端连接。

链路建立成功之后，服务端需要接收客户端的请求消息，在代码第19行我们创建新的ByteBuffer，预分配1M的缓冲区。第20行我们通过调用AsynchronousSocketChannel的read方法进行异步读操作。下面我们看看异步read方法的参数。

- ByteBuffer dst: 接收缓冲区，用于从异步Channel中读取数据包；
- A attachment: 异步Channel携带的附件，通知回调的时候作为入参使用；
- CompletionHandler<Integer,? super A>: 接收通知回调的业务handler，本例程中为ReadCompletionHandler。

下面我们继续对ReadCompletionHandler进行分析。

代码清单2-14 AIO时间服务器服务端ReadCompletionHandler

```
8.  
9.    /**  
10.    * @author lilinfeng
```

```
11.      * @date 2014年2月16日
12.      * @version 1.0
13.      */
14.      public class ReadCompletionHandler implements
15.          CompletionHandler<Integer, ByteBuffer> {
16.
17.          private AsynchronousSocketChannel channel;
18.
19.          public ReadCompletionHandler(AsynchronousSocketChan
20.              if (this.channel == null)
21.                  this.channel = channel;
22.          }
23.
24.          @Override
25.          public void completed(Integer result, ByteBuffer at
26.              attachment.flip();
27.              byte[] body = new byte[attachment.remaining()];
28.              attachment.get(body);
29.              try {
30.                  String req = new String(body, "UTF-8");
31.                  System.out.println("The time server receive ord
32.                      String currentTime = "QUERY TIME ORDER".equalsI
33.                          System.currentTimeMillis()).toString() : "B
34.                          doWrite(currentTime);
35.              } catch (UnsupportedEncodingException e) {
36.                  e.printStackTrace();
37.              }
```



```
65.         }
66.
67.         @Override
68.         public void failed(Throwable exc, ByteBuffer attach
69.         try {
70.             this.channel.close();
71.         } catch (IOException e) {
72.             e.printStackTrace();
73.         }
74.     }
75. }
```

首先看构造方法，我们将`AsynchronousSocketChannel`通过参数传递到`ReadCompletion Handler`中当作成员变量来使用，主要用于读取半包消息和发送应答。本例程不对半包读写进行具体说明，对此感兴趣的读者可以关注后续章节对`Netty`半包处理的专题介绍。我们继续看代码，第25~38行是读取到消息后的处理，首先对`attachment`进行`flip`操作，为后续从缓冲区读取数据做准备。根据缓冲区的可读字节数创建`byte`数组，然后通过`new String`方法创建请求消息，对请求消息进行判断，如果是"`QUERY TIME ORDER`"则获取当前系统服务器的时间，调用`doWrite`方法发送给客户端。下面我们对`doWrite`方法进行详细分析。

跳到代码第41行，首先对当前时间进行合法性校验，如果合法，调用字符串的解码方法将应答消息编码成字节数组，然后将它复制到发送缓冲区`writeBuffer`中，最后调用`AsynchronousSocketChannel`的异步`write`方法。正如前面介绍的异步`read`方法一样，它也有三个与`read`方法相同的参数，在本例程中我们直接实现`write`方法的异步回调接口

CompletionHandler。代码跳到第51行，对发送的writeBuffer进行判断，如果还有剩余的字节可写，说明没有发送完成，需要继续发送，直到发送成功。

最后，我们关注下failed方法，它的实现很简单，就是当发生异常的时候，对异常Throwable进行判断，如果是I/O异常，就关闭链路，释放资源，如果是其他异常，按照业务自己的逻辑进行处理。本例程作为简单demo，没有对异常进行分类判断，只要发生了读写异常，就关闭链路，释放资源。

异步非阻塞I/O版本的时间服务器服务端已经介绍完毕，下面我们继续看客户端的实现。

2.4.2 AIO创建的TimeClient源码分析

首先看下客户端主函数的实现。

代码清单2-15 AIO时间服务器客户端 TimeClient

```
16.         try {
17.             port = Integer.valueOf(args[0]);
18.         } catch (NumberFormatException e) {
19.             // 采用默认值
20.         }
21.     }
22.     new Thread(new AsyncTimeClientHandler("127.0.0.1",
23.         "AIO-AsyncTimeClientHandler-001")).start();
24.
```

```
25.      }  
26.  }
```

第22行我们通过一个独立的I/O线程创建异步时间服务器客户端handler，在实际项目中，我们不需要独立的线程创建异步连接对象，因为底层都是通过JDK的系统回调实现的，在后面运行时间服务器程序的时候，我们会抓取线程调用堆栈给大家展示。

继续看代码，AsyncTimeClientHandler的实现类源码如下。

代码清单2-16 AIO时间服务器客户端AsyncTimeClientHandler

```
1.  package com.phei.netty.aio;  
2.  
3.  import java.io.IOException;  
4.  import java.io.UnsupportedEncodingException;  
5.  import java.net.InetSocketAddress;  
6.  import java.nio.ByteBuffer;  
7.  import java.nio.channels.AsynchronousSocketChannel;  
8.  import java.nio.channels.CompletionHandler;  
9.  import java.util.concurrent.CountDownLatch;  
10.  
11. /**  
12.  * @author Administrator  
13.  * @date 2014年2月16日  
14.  * @version 1.0  
15.  */
```

```
16. public class AsyncTimeClientHandler implements
17.     CompletionHandler<Void, AsyncTimeClientHandler>, Run
18.
19.     private AsynchronousSocketChannel client;
20.     private String host;
21.     private int port;
22.     private CountdownLatch latch;
23.
24.     public AsyncTimeClientHandler(String host, int port)
25.     this.host = host;
26.     this.port = port;
27.     try {
28.         client = AsynchronousSocketChannel.open();
29.     } catch (IOException e) {
30.         e.printStackTrace();
31.     }
32. }
33.
34. @Override
35. public void run() {
36.     latch = new CountdownLatch(1);
37.     client.connect(new InetSocketAddress(host, port), th
38.     try {
39.         latch.await();
40.     } catch (InterruptedException e1) {
41.         e1.printStackTrace();
42.     }
```

```

43.         try {
44.             client.close();
45.         } catch (IOException e) {
46.             e.printStackTrace();
47.         }
48.     }
49.
50.     @Override
51.     public void completed(Void result, AsyncTimeClientHa
52.     byte[] req = "QUERY TIME ORDER".getBytes();
53.     ByteBuffer writeBuffer = ByteBuffer.allocate(req.len
54.     writeBuffer.put(req);
55.     writeBuffer.flip();
56.     client.write(writeBuffer, writeBuffer,
57.         new CompletionHandler<Integer, ByteBuffer>() {
58.             @Override
59.             public void completed(Integer result, ByteBu
60.             if (buffer.hasRemaining()) {
61.                 client.write(buffer, buffer, this);
62.             } else {
63.                 ByteBuffer readBuffer = ByteBuffer.alloc
64.                 client.read(
65.                     readBuffer,
66.                     readBuffer,
67.                     new CompletionHandler<Integer, ByteB
68.                 @Override
69.                 public void completed(Integer result

```



```

70.         ByteBuffer buffer) {
71.         buffer.flip();
72.         byte[] bytes = new byte[buffer
73.             .remaining()];
74.         buffer.get(bytes);
75.         String body;
76.         try {
77.             body = new String(bytes,
78.                 "UTF-8");
79.             System.out.println("Now is : "
80.                 + body);
81.             latch.countDown();
82.         } catch (UnsupportedEncodingException e) {
83.             e.printStackTrace();
84.         }
85.     }
86.
87.     @Override
88.     public void failed(Throwable exc,
89.         ByteBuffer attachment) {
90.         try {
91.             client.close();
92.             latch.countDown();
93.         } catch (IOException e) {
94.             // ignore on close
95.         }
96.     }

```

```

97.                });
98.            }
99.        }
100.
101.        @Override
102.        public void failed(Throwable exc, ByteBuff
103.        try {
104.            client.close();
105.            latch.countDown();
106.        } catch (IOException e) {
107.            // ignore on close
108.        }
109.    }
110.    });
111.}
112.
113.    @Override
114.    public void failed(Throwable exc, AsyncTimeClient
115.    exc.printStackTrace();
116.    try {
117.        client.close();
118.        latch.countDown();
119.    } catch (IOException e) {
120.        e.printStackTrace();
121.    }
122.    }
123.}

```

由于在AsyncTimeClientHandler中大量使用了内部匿名类，所以代码看起来稍微有些复杂，下面我们就对主要代码进行详细讲解。

24~32行是构造方法，首先通过AsynchronousSocketChannel的open方法创建一个新的AsynchronousSocketChannel对象。然后跳到第36行，创建CountDownLatch进行等待，防止异步操作没有执行完成线程就退出。第37行通过connect方法发起异步操作，它有两个参数，分别如下。

- **A attachment:** AsynchronousSocketChannel的附件，用于回调通知时作为入参被传递，调用者可以自定义；
- **CompletionHandler<Void,? super A> handler:** 异步操作回调通知接口，由调用者实现。

在本例程中，我们的两个参数都使用AsyncTimeClientHandler类本身，因为它实现了CompletionHandler接口。

接下来我们看异步连接成功之后的方法回调——completed方法。代码第52行，我们创建请求消息体，对其进行编码，然后复制到发送缓冲区writeBuffer中，调用Asynchronous SocketChannel的write方法进行异步写。与服务端类似，我们可以实现CompletionHandler <Integer, ByteBuffer>接口用于写操作完成后的回调。代码第60~62行，如果发送缓冲区中仍有尚未发送的字节，将继续异步发送，如果已经发送完成，则执行异步读取操作。

代码第64~97行是客户端异步读取时间服务器服务端应答消息的处理逻辑。代码第64行调用AsynchronousSocketChannel的read方法异步读取服务端的响应消息。由于read操作是异步的，所以我们通过内部匿名

类实现CompletionHandler<Integer, ByteBuffer>接口，当读取完成被JDK回调时，构造应答消息。第71~78行从CompletionHandler的ByteBuffer中读取应答消息，然后打印结果。

第102~111行，当读取发生异常时，关闭链路，同时调用CountDownLatch的countDown方法让AsyncTimeClientHandler线程执行完毕，客户端退出执行。

需要指出的是，正如之前的NIO例程，我们并没有完整的处理网络的半包读写，在对例程进行功能测试的时候没有问题，但是，如果对代码稍加改造，进行压力或者性能测试，就会发现输出结果存在问题。

由于半包读写会作为专门的小节在Netty的应用和源码分析章节进行详细讲解，在NIO的入门章节我们就不详细展开介绍了，以便读者能够将注意力集中在NIO的入门知识上来。

在下面的小节中我们会运行AIO版本的时间服务器程序，并通过打印线程堆栈的方式看下JDK回调异步Channel CompletionHandler的调用情况。

2.4.3 AIO版本时间服务器运行结果

执行TimeServer，运行结果如图2-15所示。

图2-15 AIO时间服务器服务端运行结果

执行TimeClient，运行结果如图2-16所示。

图2-16 AIO时间服务器客户端运行结果

下面我们继续看下JDK异步回调CompletionHandler的线程执行堆栈。

图2-17 AIO时间服务器异步回调线程堆栈

从“Thread-2”线程堆栈中可以发现，JDK底层通过线程池ThreadPoolExecutor来执行回调通知，异步回调通知类由sun.nio.ch.AsynchronousChannelGroupImpl实现，它经过层层调用，最终回调com.phei.netty.aio.AsyncTimeClientHandler\$1.completed方法，完成回调通知。由此我们也可以得出结论：异步Socket Channel是被动执行对象，我们不需要像NIO编程那样创建一个独立的I/O线程来处理读写操作。对于AsynchronousServerSocket Channel和AsynchronousSocketChannel，它们都由JDK底层的线程池负责回调并驱动读写操作。正因为如此，基于NIO2.0新的异步非阻塞Channel进行编程比NIO编程更为简单。

本小节我们讲解了JDK1.7提供的新的异步非阻塞I/O（AIO）的用法，由于国内商用的主流Java版本仍然是JDK1.6，因此，本小节不再详细介绍NIO2.0其他新增的特性，如果大家对NIO2.0的异步文件操作等特性感兴趣，可以选择阅读JDK1.7的相关书籍或者查看甲骨文发布的JDK1.7白皮书。

下个小节我们对本章列举的5种I/O进行概念澄清和比较，让大家从整体上掌握这些I/O模型的差异，以便在未来的工作中能够根据产品的实际情况选择合适的I/O模型。

2.5 4种I/O的对比

2.5.1 概念澄清

为了防止由于对一些技术概念和术语的理解或者叫法不一致而引起歧义，本小节特意对本书中的专业术语或者技术用语做下声明：如果它们与其他一些技术书籍的称呼不一致，请以本小节的解释为准。

1. 异步非阻塞I/O

很多人喜欢将JDK1.4提供的NIO框架称为异步非阻塞I/O，但是，如果严格按照UNIX网络编程模型和JDK的实现进行区分，实际上它只能被称为非阻塞I/O，不能叫异步非阻塞I/O。在早期的JDK1.4和1.5 update10版本之前，JDK的Selector基于select/poll模型实现，它是基于I/O复用技术的非阻塞I/O，不是异步I/O。在JDK1.5 update10和Linux core2.6以上版本，Sun优化了Selector的实现，它在底层使用epoll替换了select/poll，上层的API并没有变化，可以认为是JDK NIO的一次性能优化，但是它仍旧没有改变I/O的模型。相关优化的官方说明如图2-17所示。

由JDK1.7提供的NIO2.0，新增了异步的套接字通道，它是真正的异步I/O，在异步I/O操作的时候可以传递信号变量，当操作完成之后会回调相关的方法，异步I/O也被称为AIO。

NIO类库支持非阻塞读和写操作，相比于之前的同步阻塞读和写，它是异步的，因此很多人习惯于称NIO为异步非阻塞I/O，包括很多介绍NIO编程的书籍也沿用了这个说法。为了符合大家的习惯，本书也会将

NIO称为异步非阻塞I/O或者非阻塞I/O，请大家理解，不要过分纠结在一些技术术语的咬文嚼字上。

图2-17 JDK1.5_update10支持epoll

2. 多路复用器Selector

几乎所有的中文技术书籍都将Selector翻译为选择器，但是实际上我认为这样的翻译并不恰当，选择器仅仅是字面上的意思，体现不出Selector的功能和特点。

在前面的章节我们介绍过Java NIO的实现关键是多路复用I/O技术，多路复用的核心就是通过Selector来轮询注册在其上的Channel，当发现某个或者多个Channel处于就绪状态后，从阻塞状态返回就绪的Channel的选择键集合，进行I/O操作。由于多路复用器是NIO实现非阻塞I/O的关键，它又是主要通过Selector实现的，所以本书将Selector翻译为多路复用器，与其他技术书籍所说的选择器是同一个东西，请大家了解。

3. 伪异步I/O

伪异步I/O的概念完全来源于实践。在JDK NIO编程没有流行之前，为了解决Tomcat通信线程同步I/O导致业务线程被挂住的问题，大家想到了一个办法：在通信线程和业务线程之间做个缓冲区，这个缓冲区用于隔离I/O线程和业务线程间的直接访问，这样业务线程就不会被I/O线程阻塞。而对于后端的业务侧来说，将消息或者Task放到线程池后就返回了，它不再直接访问I/O线程或者进行I/O读写，这样也就不会被同步阻塞。类似的设计还包括前端启动一组线程，将接收的客户端封装成Task，放到后端的线程池执行，用于解决一连接一线程问题。像这样通

过线程池做缓冲区的做法，本书中习惯于称它为伪异步I/O，而官方并没有伪异步I/O这种说法，请大家注意。

下面的小节我们对几种常见的I/O进行对比，以便大家能够理解几种I/O的差异。

2.5.2 不同I/O模型对比

不同的I/O模型由于线程模型、API等差别很大，所以用法的差异也非常大。由于之前的几个小节已经集中对这几种I/O的API和用法进行了说明，本小节会重点对这几种I/O进行功能对比。如表2-1所示。

表2-1 几种I/O模型的功能和特性对比

尽管本书是专门介绍NIO框架Netty的，但是，并不意味着所有的Java网络编程都必须要选择NIO和Netty，具体选择什么样的I/O模型或者NIO框架，完全基于业务的实际应用场景和性能诉求，如果客户端并发连接数不多，周边对接的网元不多，服务器的负载也不重，那就完全没必要选择NIO做服务端；如果是相反情况，那就要考虑选择合适的NIO框架进行开发。

对比完Java的几种主流I/O模型之后，我们继续看下为什么要选择Netty进行NIO开发，而不是直接使用JDK的NIO原生类库。

2.6 选择Netty的理由

在开始本节之前，我先讲一个亲身经历的故事：曾经有两个项目组同时用到了NIO编程技术，一个项目组选择自己开发NIO服务端，直接使用JDK原生的API，结果两个多月过去了，他们的NIO服务端始终无法稳定，问题频出。由于NIO通信是它们的核心组件之一，因此，项目的进度受到了严重的影响，领导对此也非常恼火。另一个项目组直接使用Netty作为NIO服务端，业务的定制开发工作量非常小，测试表明，功能和性能都完全达标，项目组几乎没有在NIO服务端上花费额外的时间和精力，项目进展也非常顺利。

这两个项目组的不同遭遇告诉我们：开发出高质量的NIO程序并不是一件简单的事情，除去NIO固有的复杂性和BUG不谈，作为一个NIO服务端，需要能够处理网络的闪断、客户端的重复接入、客户端的安全认证、消息的编解码、半包读写等情况，如果你没有足够的NIO编程经验积累，一个NIO框架的稳定往往需要半年甚至更长的时间。更为糟糕的是，一旦在生产环境中发生问题，往往会导致跨节点的服务调用中断，严重的可能会导致整个集群环境都不可用，需要重启服务器，这种非正常停机会带来巨大的损失。

从可维护性角度看，由于NIO采用了异步非阻塞编程模型，而且是一个I/O线程处理多条链路，它的调试和跟踪非常麻烦，特别是生产环境中的问题，我们无法进行有效的调试和跟踪，往往只能靠一些日志来辅助分析，定位难度很大。

2.6.1 不选择Java原生NIO编程的原因

现在我们总结一下为什么不建议开发者直接使用JDK的NIO类库进行开发，具体原因如下。

(1) NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。

(2) 需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程涉及到Reactor模式，你必须对多线程和网络编程非常熟悉，才能编写出高质量的NIO程序。

(3) 可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。

(4) JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有被根本解决。该BUG以及与该BUG相关的问题单可以参见以下链接内容。

- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933
- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2147719

异常堆栈如下。

```
java.lang.Thread.State: RUNNABLE

at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
```

at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:210)

at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:6

at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)

```
- locked <0x00000000750928190> (a sun.nio.ch.Util$2)
- locked <0x000000007509281a8> (a java.util.Collection
- locked <0x00000000750946098> (a sun.nio.ch.EPollSele
```

at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:80)

```
at net.spy.memcached.MemcachedConnection.handleIO(Mem
at net.spy.memcached.MemcachedConnection.run(Memcache
```

由于上述原因，在大多数场景下，不建议大家直接使用JDK的NIO类库，除非你精通NIO编程或者有特殊的需求。在绝大多数的业务场景中，我们可以使用NIO框架Netty来进行NIO编程，它既可以作为客户端也可以作为服务端，同时支持UDP和异步文件传输，功能非常强大。

下个小节我们就看看为什么选择Netty作为基础通信框架。

2.6.2 为什么选择Netty

Netty是业界最流行的NIO框架之一，它的健壮性、功能、性能、可

定制性和可扩展性在同类框架中都是首屈一指的，它已经得到成百上千的商用项目验证，例如Hadoop的RPC框架avro使用Netty作为底层通信框架；很多其他业界主流的RPC框架，也使用Netty来构建高性能的异步通信能力。

通过对Netty的分析，我们将它的优点总结如下。

- API使用简单，开发门槛低；
- 功能强大，预置了多种编解码功能，支持多种主流协议；
- 定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；
- 性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；
- 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼；
- 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；
- 经历了大规模的商业应用考验，质量得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它已经完全能够满足不同行业的商业应用了。

正是因为这些优点，Netty逐渐成为Java NIO编程的首选框架。

2.7 总结

本章通过一个简单的demo开发，即时间服务器程序，让大家熟悉传统的同步阻塞I/O、伪异步I/O、非阻塞I/O（NIO）和异步I/O（AIO）的编程和使用差异，然后对比了各自的优缺点，并给出了使用建议。

最后，我们详细介绍了为什么不建议读者朋友们直接使用JDK的NIO原生类库进行异步I/O的开发，同时对Netty的优点进行分析和总结，给出使用Netty进行NIO开发的理由。

相信学完本章之后，大家对Java的网络编程已经有了初步的认识，从下一个章节开始，我们正式进入Netty的世界，学习和掌握基于Netty的网络开发。

入门篇 **Netty NIO**开发指南

第3章 Netty入门应用

第4章 TCP粘包/拆包问题的解决之道

第5章 分隔符和定长解码器的应用

第3章 Netty入门应用

作为Netty的第一个应用程序，我们依然以第2章的时间服务器为例进行开发，通过Netty版本的时间服务器的开发，让初学者尽快学到如何搭建Netty开发环境和运行Netty应用程序。

如果你已经熟悉Netty的基础应用，可以跳过本章，继续后面知识的学习。

本章主要包括：

- Netty开发环境的搭建
- 服务端程序TimeServer开发
- 客户端程序TimeClient开发
- 时间服务器的运行和调试

3.1 Netty开发环境的搭建

首先假设你已经在本机安装了JDK1.7，配置了JDK的环境变量path，同时下载并正确启动了IDE工具Eclipse。如果你是个Java初学者，从来没有在本机搭建过Java开发环境，建议你先选择一本Java基础入门的书籍或者课程进行学习。

假如你习惯于使用其他IDE工具进行Java开发，例如NetBeans IDE，也可以运行本节的入门例程。但是，你需要根据自己实际使用的IDE进行对应的配置修改和调整，本书统一使用eclipse-jee-kepler-SR1-win32作为Java开发工具。

下面我们开始学习如何搭建Netty的开发环境。

3.1.1 下载Netty的软件包

访问Netty的官网<http://netty.io/>，从【Downloads】标签页选择下载5.0.0.Alpha1安装包，安装包不大，8.95M左右，下载之后的安装包如图3-1所示。

图3-1 Netty 5.0压缩包

通过解压缩工具打开压缩包，目录如图3-2所示。

图3-2 Netty 5.0压缩包内部目录

这时会发现里面包含了各个模块的.jar包和源码，由于我们直接以二进制类库的方式使用Netty，所以只需要获取netty-all-5.0.0.Alpha1.jar即可。

3.1.2 搭建Netty应用工程

使用Eclipse创建普通的Java工程，同时创建Java源文件的package，如图3-3所示。

图3-3 Netty应用工程

创建第三方类库存放文件夹lib，同时将netty-all-5.0.0.Alpha1.jar复制到lib目录下，如图3-4所示。

图3-4 配置引用的netty jar包

右键单击netty-all-5.0.0.Alpha1.jar，在弹出的菜单中，选择将.jar包添加到Build Path中，操作如图3-5所示。

图3-5 将Netty.jar包添加到ClassPath中

到此结束，我们的Netty应用开发环境已经搭建完成，下面的小节将演示如何基于Netty开发时间服务器程序。

3.2 Netty服务端开发

作为第一个Netty的应用例程，为了让读者能够将精力集中在Netty的使用上，我们依然选择第2章的时间服务器为例进行源码开发和代码讲解。

TimeServer开发

在开始使用Netty开发TimeServer之前，先回顾一下使用NIO进行服务端开发的步骤。

- (1) 创建ServerSocketChannel，配置它为非阻塞模式；
- (2) 绑定监听，配置TCP参数，例如backlog大小；
- (3) 创建一个独立的I/O线程，用于轮询多路复用器Selector；
- (4) 创建Selector，将之前创建的ServerSocketChannel注册到Selector上，监听SelectionKey.ACCEPT；
- (5) 启动I/O线程，在循环体中执行Selector.select()方法，轮询就绪的Channel；
- (6) 当轮询到了处于就绪状态的Channel时，需要对其进行判断，如果是OP_ACCEPT状态，说明是新的客户端接入，则调用ServerSocketChannel.accept()方法接受新的客户端；
- (7) 设置新接入的客户端链路SocketChannel为非阻塞模式，配置其他的一些TCP参数；

(8) 将SocketChannel注册到Selector，监听OP_READ操作位；

(9) 如果轮询的Channel为OP_READ，则说明SocketChannel中有新的就绪的数据包需要读取，则构造ByteBuffer对象，读取数据包；

(10) 如果轮询的Channel为OP_WRITE，说明还有数据没有发送完成，需要继续发送。

一个简单的NIO服务端程序，如果我们直接使用JDK的NIO类库进行开发，竟然需要经过烦琐的十多步操作才能完成最基本的消息读取和发送，这也是我们要选择Netty等NIO框架的原因了，下面我们看看使用Netty是如何轻松搞定服务端开发的。

代码清单3-1 Netty时间服务器服务端TimeServer

```
16. public class TimeServer {
17.
18.     public void bind(int port) throws Exception {
19.         // 配置服务端的NIO线程组
20.         EventLoopGroup bossGroup = new NioEventLoopGroup();
21.         EventLoopGroup workerGroup = new NioEventLoopGroup(
22.             try {
23.                 ServerBootstrap b = new ServerBootstrap();
24.                 b.group(bossGroup, workerGroup)
25.                     .channel(NioServerSocketChannel.class)
26.                     .option(ChannelOption.SO_BACKLOG, 1024)
27.                     .childHandler(new ChildChannelHandler());
28.                 // 绑定端口，同步等待成功
```

```
29.         ChannelFuture f = b.bind(port).sync();
30.
31.         // 等待服务端监听端口关闭
32.         f.channel().closeFuture().sync();
33.     } finally {
34.         // 优雅退出，释放线程池资源
35.         bossGroup.shutdownGracefully();
36.         workerGroup.shutdownGracefully();
37.     }
38. }
39.
40. private class ChildChannelHandler extends ChannelIn
41. @Override
42. protected void initChannel(SocketChannel arg0) thro
43.     arg0.pipeline().addLast(new TimeServerHandler()
44. }
45.
46. }
47.
48. /**
49.  * @param args
50.  * @throws Exception
51.  */
52. public static void main(String[] args) throws Excep
53. int port = 8080;
54. if (args != null && args.length > 0) {
55.     try {
```

```
56.         port = Integer.valueOf(args[0]);
57.     } catch (NumberFormatException e) {
58.         // 采用默认值
59.     }
60. }
61. new TimeServer().bind(port);
62. }
63. }
```

由于本章的重点是讲解Netty的应用开发，所以对于一些Netty的类库和用法仅仅做基础性的讲解，我们从黑盒的角度理解这些概念即可。后续源码分析章节会专门对Netty核心的类库和功能进行分析，感兴趣的同学可以跳到源码分析章节进行后续的学习。

我们从bind方法开始学习，在代码第20～21行创建了两个NioEventLoopGroup实例。NioEventLoopGroup是个线程组，它包含了一组NIO线程，专门用于网络事件的处理，实际上它们就是Reactor线程组。这里创建两个的原因是一个用于服务端接受客户端的连接，另一个用于进行SocketChannel的网络读写。第23行我们创建ServerBootstrap对象，它是Netty用于启动NIO服务端的辅助启动类，目的是降低服务端的开发复杂度。第24行调用ServerBootstrap的group方法，将两个NIO线程组当作入参传递到ServerBootstrap中。接着设置创建的Channel为NioServerSocketChannel，它的功能对应于JDK NIO类库中的ServerSocketChannel类。然后配置NioServerSocketChannel的TCP参数，此处将它的backlog设置为1024，最后绑定I/O事件的处理类ChildChannelHandler，它的作用类似于Reactor模式中的handler类，主要用于处理网络I/O事件，例如记录日志、对消息进行编解码等。

服务端启动辅助类配置完成之后，调用它的bind方法绑定监听端口，随后，调用它的同步阻塞方法sync等待绑定操作完成。完成之后Netty会返回一个ChannelFuture，它的功能类似于JDK的java.util.concurrent.Future，主要用于异步操作的通知回调。

第32行使用f.channel().closeFuture().sync()方法进行阻塞，等待服务端链路关闭之后main函数才退出。

第34~36行调用NIO线程组的shutdownGracefully进行优雅退出，它会释放跟shutdownGracefully相关联的资源。

下面看看TimeServerHandler类是如何实现的。

代码清单3-2 Netty时间服务器服务端TimeServerHandler

```
12. public class TimeServerHandler extends ChannelHandlerAd
13.
14.     @Override
15.     public void channelRead(ChannelHandlerContext ctx,
16.         throws Exception {
17.         ByteBuf buf = (ByteBuf) msg;
18.         byte[] req = new byte[buf.readableBytes()];
19.         buf.readBytes(req);
20.         String body = new String(req, "UTF-8");
21.         System.out.println("The time server receive order :
22.         String currentTime = "QUERY TIME ORDER".equalsIgnoreCase
23.             System.currentTimeMillis()).toString() : "BAD O
24.         ByteBuf resp = Unpooled.copiedBuffer(currentTime.ge
```

```
25.         ctx.write(resp);
26.     }
27.
28.     @Override
29.     public void channelReadComplete(ChannelHandlerContext c
30.         ctx.flush();
31.     }
32.
33.     @Override
34.     public void exceptionCaught(ChannelHandlerContext c
35.         ctx.close();
36.     }
37. }
```

`TimeServerHandler`继承自`ChannelHandlerAdapter`，它用于对网络事件进行读写操作，通常我们只需要关注`channelRead`和`exceptionCaught`方法。下面对这两个方法进行简单说明。

第17行做类型转换，将`msg`转换成Netty的`ByteBuf`对象。`ByteBuf`类似于JDK中的`java.nio.ByteBuffer` 对象，不过它提供了更加强大和灵活的功能。通过`ByteBuf`的`readableBytes`方法可以获取缓冲区可读的字节数，根据可读的字节数创建`byte`数组，通过`ByteBuf`的`readBytes`方法将缓冲区中的字节数组复制到新建的`byte`数组中，最后通过`new String`构造函数获取请求消息。这时对请求消息进行判断，如果是"`QUERY` `TIME ORDER`"则创建应答消息，通过`ChannelHandlerContext`的`write`方法异步发送应答消息给客户端。

第30行我们发现还调用了ChannelHandlerContext的flush方法，它的作用是将消息发送队列中的消息写入到SocketChannel中发送给对方。从性能角度考虑，为了防止频繁地唤醒Selector进行消息发送，Netty的write方法并不直接将消息写入SocketChannel中，调用write方法只是把待发送的消息放到发送缓冲数组中，再通过调用flush方法，将发送缓冲区中的消息全部写到SocketChannel中。

第35行，当发生异常时，关闭ChannelHandlerContext，释放和ChannelHandlerContext相关联的句柄等资源。

通过对代码进行统计分析可以看出，不到30行的业务逻辑代码，即完成了NIO服务端的开发，相比于传统基于JDK NIO原生类库的服务端，代码量大大减少，开发难度也降低了很多。

下面我们继续学习客户端的开发，并使用Netty改造TimeClient。

3.3 Netty客户端开发

Netty客户端的开发相比于服务端更简单，下面我们就看下客户端的代码如何实现。

TimeClient开发

代码清单3-3 Netty时间服务器客户端TimeClient

```
16. public class TimeClient {
17.
18.     public void connect(int port, String host) throws E
19.         // 配置客户端NIO线程组
20.         EventLoopGroup group = new NioEventLoopGroup();
21.         try {
22.             Bootstrap b = new Bootstrap();
23.             b.group(group).channel(NioSocketChannel.class)
24.                 .option(ChannelOption.TCP_NODELAY, true)
25.                 .handler(new ChannelInitializer<SocketChann
26.                     @Override
27.                     public void initChannel(SocketChannel ch)
28.                         throws Exception {
29.                         ch.pipeline().addLast(new TimeClientHan
30.                     }
31.                 });
32.
```

```
33.         // 发起异步连接操作
34.         ChannelFuture f = b.connect(host, port).sync();
35.
36.         // 等待客户端链路关闭
37.         f.channel().closeFuture().sync();
38.     } finally {
39.         // 优雅退出，释放NIO线程组
40.         group.shutdownGracefully();
41.     }
42. }
43.
44. /**
45.  * @param args
46.  * @throws Exception
47.  */
48. public static void main(String[] args) throws Excep
49. int port = 8080;
50. if (args != null && args.length > 0) {
51.     try {
52.         port = Integer.valueOf(args[0]);
53.     } catch (NumberFormatException e) {
54.         // 采用默认值
55.     }
56. }
57. new TimeClient().connect(port, "127.0.0.1");
58. }
59. }
```

我们从connect方法讲起，在第20行首先创建客户端处理I/O读写的NioEventLoopGroup线程组，然后继续创建客户端辅助启动类Bootstrap，随后需要对其进行配置。与服务端不同的是，它的Channel需要设置为NioSocketChannel，然后为其添加handler，此处为了简单直接创建匿名内部类，实现initChannel方法，其作用是当创建NioSocketChannel成功之后，在初始化它的时候将它的ChannelHandler设置到ChannelPipeline中，用于处理网络I/O事件。

客户端启动辅助类设置完成之后，调用connect方法发起异步连接，然后调用同步方法等待连接成功。

最后，当客户端连接关闭之后，客户端主函数退出，在退出之前，释放NIO线程组的资源。

下面我们继续看下TimeClientHandler的代码如何实现。

代码清单3-4 Netty时间服务器客户端TimeClientHandler

```
14. public class TimeClientHandler extends ChannelHandlerAd
15.
16.     private static final Logger logger = Logger
17.         .getLogger(TimeClientHandler.class.getName());
18.
19.     private final ByteBuf firstMessage;
20.
21.     /**
22.      * Creates a client-side handler.
```

```
23.         */
24.     public TimeClientHandler() {
25.         byte[] req = "QUERY TIME ORDER".getBytes();
26.         firstMessage = Unpooled.buffer(req.length);
27.         firstMessage.writeBytes(req);
28.
29.     }
30.
31.     @Override
32.     public void channelActive(ChannelHandlerContext ctx
33.         ctx.writeAndFlush(firstMessage);
34.     }
35.
36.     @Override
37.     public void channelRead(ChannelHandlerContext ctx,
38.         throws Exception {
39.         ByteBuf buf = (ByteBuf) msg;
40.         byte[] req = new byte[buf.readableBytes()];
41.         buf.readBytes(req);
42.         String body = new String(req, "UTF-8");
43.         System.out.println("Now is : " + body);
44.     }
45.
46.     @Override
47.     public void exceptionCaught(ChannelHandlerContext c
48.         // 释放资源
49.         logger.warning("Unexpected exception from downstrea
```

```
50.         + cause.getMessage());  
51.     ctx.close();  
52. }  
53. }
```

这里重点关注三个方法：`channelActive`、`channelRead`和`exceptionCaught`。当客户端和服务端TCP链路建立成功之后，Netty的NIO线程会调用`channelActive`方法，发送查询时间的指令给服务端，调用`ChannelHandlerContext`的`writeAndFlush`方法将请求消息发送给服务端。

当服务端返回应答消息时，`channelRead`方法被调用，第39～43行从Netty的`ByteBuf`中读取并打印应答消息。

第47～52行，当发生异常时，打印异常日志，释放客户端资源。

3.4 运行和调试

3.4.1 服务端和客户端的运行

在Eclipse开发环境中运行和调试Java程序非常简单，下面我们看下如何运行TimeServer：将光标定位到TimeServer类中，单击右键，在弹出菜单中选择Run As → Java Application，或者直接使用快捷键Alt+Shift+X执行，如图3-6所示。

图3-6 运行TimeServer

客户端的执行类似，可以看到以下执行结果。

服务端运行结果如图3-7所示。

图3-7 TimeServer运行结果

客户端运行结果如图3-8所示。

图3-8 TimeClient运行结果

运行结果正确。可以发现，通过Netty开发的NIO服务端和客户端非常简单，短短几十行代码，就能完成之前NIO程序需要几百行才能完成的功能。基于Netty的应用开发不但API使用简单、开发模式固定，而且扩展性和定制性非常好，后面，我们会通过更多应用来介绍Netty的强大功能。

需要指出的是，本例程依然没有考虑读半包的处理，对于功能演示或者测试，上述程序没有问题，但是稍加改造进行性能或者压力测试，

它就不能正确地工作了。在下一个章节我们会给出能够正确处理半包消息的应用实例。

3.4.2 打包和部署

基于Netty开发的都是非Web的Java应用，它的打包形态非常简单，就是一个普通的.jar包，通常情况下，在正式的商业开发中，我们会使用三种打包方式对源码进行打包：

（1）Eclipse提供的导出功能。它可以将指定的Java工程或者源码包、代码输出成指定的.jar包，它属于手工操作，当项目模块多之后非常不方便，所以一般不使用这种方式；

（2）使用ant脚本对工程进行打包。将Netty的应用程序打包成指定的.jar包，一般会输出一个软件安装包：xxxx_install.gz；

（3）使用Maven进行工程构建。它可以对模块间的依赖进行管理，支持版本的自动化测试、编译和构建，是目前主流的项目管理工具。

3.5 总结

本章节讲解了Netty的入门应用，通过使用Netty重构时间服务器程序，可以发现相比于传统的NIO程序，Netty的代码更加简洁、开发难度更低，扩展性也更好，非常适合作为基础通信框架被用户集成和使用。

在介绍Netty服务端和客户端时，简单地对代码进行了讲解，由于后续会有专门章节对Netty进行源码分析，所以在Netty应用部分我们不进行详细的源码解读和分析。

第4章会讲解一个稍微复杂的应用，它利用Netty提供的默认编解码功能解决了我们之前没有解决的读半包问题。事实上，对于读半包问题，Netty提供了很多种好的解决方案。下面一起学习一下如何利用Netty默认的编解码功能解决半包读取问题。

第4章 TCP粘包/拆包问题的解决之道

熟悉TCP编程的读者可能都知道，无论是服务端还是客户端，当我们读取或者发送消息的时候，都需要考虑TCP底层的粘包/拆包机制。本章开始我们先简单介绍TCP粘包/拆包的基础知识，然后模拟一个没有考虑TCP粘包/拆包导致功能异常的案例，最后，通过正确例程来探讨Netty是如何解决这个问题的。

如果你已经熟悉了TCP粘包和拆包的相关知识，建议你直接跳到代码讲解小节，看Netty是如何解决这个问题的。

本章主要包括：

- TCP粘包/拆包的基础知识
- 没考虑TCP粘包/拆包的问题案例
- 使用Netty解决读半包问题

4.1 TCP粘包/拆包

TCP是个“流”协议，所谓流，就是没有界限的一串数据。大家可以想想河里的流水，是连成一片的，其间并没有分界线。TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的TCP粘包和拆包问题。

4.1.1 TCP粘包/拆包问题说明

我们可以通过图解对TCP粘包和拆包问题进行说明，粘包问题示例如图4-1所示。

图4-1 TCP粘包/拆包问题

假设客户端分别发送了两个数据包D1和D2给服务端，由于服务端一次读取到的字节数是不确定的，故可能存在以下4种情况。

（1）服务端分两次读取到了两个独立的数据包，分别是D1和D2，没有粘包和拆包；

（2）服务端一次接收到了两个数据包，D1和D2粘合在一起，被称为TCP粘包；

（3）服务端分两次读取到了两个数据包，第一次读取到了完整的D1包和D2包的部分内容，第二次读取到了D2包的剩余内容，这被称为TCP拆包；

(4) 服务端分两次读取到了两个数据包，第一次读取到了D1包的部分内容D1_1，第二次读取到了D1包的剩余内容D1_2和D2包的整包。

如果此时服务端TCP接收滑窗非常小，而数据包D1和D2比较大，很有可能会发生第五种可能，即服务端分多次才能将D1和D2包接收完全，期间发生多次拆包。

4.1.2 TCP粘包/拆包发生的原因

问题产生的原因有三个，分别如下。

- (1) 应用程序write写入的字节大小大于套接口发送缓冲区大小；
- (2) 进行MSS大小的TCP分段；
- (3) 以太网帧的payload大于MTU进行IP分片。

图解如图4-2所示。

图4-2 TCP粘包/拆包问题原因

4.1.3 粘包问题的解决策略

由于底层的TCP无法理解上层的业务数据，所以在底层是无法保证数据包不被拆分和重组的，这个问题只能通过上层的应用协议栈设计来解决，根据业界的主流协议的解决方案，可以归纳如下。

- (1) 消息定长，例如每个报文的大小为固定长度200字节，如果不够，空位补空格；
- (2) 在包尾增加回车换行符进行分割，例如FTP协议；

(3) 将消息分为消息头和消息体，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思路为消息头的第一个字段使用 `int32` 来表示消息的总长度；

(4) 更复杂的应用层协议。

介绍完了TCP粘包/拆包的基础知识，下面我们就通过实际例程来看看如何使用Netty提供的半包解码器来解决TCP粘包/拆包问题。

4.2 未考虑TCP粘包导致功能异常案例

在前面的时间服务器例程中，我们多次强调并没有考虑读半包问题，这在功能测试时往往没有问题，但是一旦压力上来，或者发送大报文之后，就会存在粘包/拆包问题。如果代码没有考虑，往往就会出现解码错位或者错误，导致程序不能正常工作。下面我们以3.1节的代码为例，模拟故障场景，然后看看如何正确使用Netty的半包解码器来解决TCP粘包/拆包问题。

4.2.1 TimeServer的改造

代码清单4-1 Netty时间服务器服务端TimeServerHandler

```
12. public class TimeServerHandler extends ChannelHandlerAdapti
13.
14.     private int counter;
15.
16.     @Override
17.     public void channelRead(ChannelHandlerContext ctx, Obje
18.         throws Exception {
19.         ByteBuf buf = (ByteBuf) msg;
20.         byte[] req = new byte[buf.readableBytes()];
21.         buf.readBytes(req);
22.         String body = new String(req, "UTF-8").substring(0, req
23.             - System.getProperty("line.separator").length());
24.         System.out.println("The time server receive order : ")
```

```

25.         + " ; the counter is : " + ++counter);
26.     String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(
27.         System.currentTimeMillis()).toString() : "BAD ORDER";
28.     currentTime = currentTime + System.getProperty("line.separator");
29.     ByteBuf resp = Unpooled.copiedBuffer(currentTime.getBytes());
30.     ctx.writeAndFlush(resp);
31. }
32.
33. @Override
34. public void exceptionCaught(ChannelHandlerContext ctx,
35.     Throwable cause) throws Exception {
36.     ctx.close();
37. }

```

每读到一条消息后，就计一次数，然后发送应答消息给客户端。按照设计，服务端接收到的消息总数应该跟客户端发送的消息总数相同，而且请求消息删除回车换行符后应该为"QUERY TIME ORDER"。下面我们继续看下客户端的改造。

4.2.2 TimeClient的改造

代码清单4-2 Netty时间服务器客户端TimeClientHandler

```

14. public class TimeClientHandler extends ChannelHandlerAdapter {
15.
16.     private static final Logger logger = Logger.getLogger(
17.         TimeClientHandler.class.getName());

```

```
18.
19.     private int counter;
20.
21.     private byte[] req;
22.
23.     /**
24.      * Creates a client-side handler.
25.      */
26.     public TimeClientHandler() {
27. req = ("QUERY TIME ORDER" + System.getProperty("line.separ
28.     .getBytes());
29.     }
30.
31.     @Override
32.     public void channelActive(ChannelHandlerContext ctx) {
33. ByteBuf message = null;
34. for (int i = 0; i < 100; i++) {
35.     message = Unpooled.buffer(req.length);
```



```
36.     message.writeBytes(req);
```

```
37.     ctx.writeAndFlush(message);
```

```
38. }
```

```
39. }
```

```
40.
```

```
41.     @Override
```

```
42.     public void channelRead(ChannelHandlerContext ctx, Object
```

```
43.         throws Exception {
```

```
44.     ByteBuf buf = (ByteBuf) msg;
```

```
45.     byte[] req = new byte[buf.readableBytes()];
```

```
46.     buf.readBytes(req);
```

```
47.     String body = new String(req, "UTF-8");
```

```
48.     System.out.println("Now is : " + body + " ; the counter is
```

```
49.         + ++counter);
```

```
50.     }
51.
52.     @Override
53.     public void exceptionCaught(ChannelHandlerContext ctx,
54.         // 释放资源
55. logger.warning("Unexpected exception from downstream : "
56.     + cause.getMessage());
57. ctx.close();
58.     }
59. }
```

主要的修改点就是代码33~38行，客户端跟服务端链路建立成功之后，循环发送100条消息，每发送一条就刷新一次，保证每条消息都会被写入Channel中。按照我们的设计，服务端应该接收到100条查询时间指令的请求消息。

第48~49行，客户端每接收到服务端一条应答消息之后，就打印一次计数器。按照设计初衷，客户端应该打印100次服务端的系统时间。

下面的小节就来看下运行结果是否符合设计初衷。

4.2.3 运行结果

分别执行服务端和客户端，运行结果如下。

服务端运行结果如下。

The time server receive order : QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORD ; the counter is : 1

The time server receive order :

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

```
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER
QUERY TIME ORDER ; the counter is : 2
```

服务端运行结果表明它只接收到了两条消息，第一条包含57条“QUERY TIME ORDER”指令，第二条包含了43条“QUERY TIME ORDER”指令，总数正好是100条。我们期待的是收到100条消息，每条包含一条“QUERY TIME ORDER”指令。这说明发生了TCP粘包。

客户端运行结果如下。

```
Now is : BAD ORDER
BAD ORDER
; the counter is : 1
```

按照设计初衷，客户端应该收到100条当前系统时间的消息，但实际上只收到了一条。这不难理解，因为服务端只收到了2条请求消息，所以实际服务端只发送了2条应答，由于请求消息不满足查询条件，所以返回了2条“BAD ORDER”应答消息。但是实际上客户端只收到了一条包含2条“BAD ORDER”指令的消息，说明服务端返回的应答消息也发生了粘包。

由于上面的例程没有考虑TCP的粘包/拆包，所以当发生TCP粘包时，我们的程序就不能正常工作。

下面的章节将演示如何通过Netty的LineBasedFrameDecoder和StringDecoder来解决TCP粘包问题。

4.3 利用**LineBasedFrameDecoder**解决TCP粘包问题

为了解决TCP粘包/拆包导致的半包读写问题，Netty默认提供了多种编解码器用于处理半包，只要能熟练掌握这些类库的使用，TCP粘包问题从此会变得非常容易，你甚至不需要关心它们，这也是其他NIO框架和JDK原生的NIO API所无法匹敌的。

下面我们就以修正时间服务器为目标进行开发和讲解，通过对实际代码的讲解让大家能够尽快熟悉和掌握半包解码器的使用。

4.3.1 支持TCP粘包的TimeServer

直接看代码，然后对LineBasedFrameDecoder和StringDecoder的API进行说明。

代码清单4-3 Netty时间服务器服务端TimeServer

```
18. public class TimeServer {
19.
20.     public void bind(int port) throws Exception {
21.         // 配置服务端的NIO线程组
22.         EventLoopGroup bossGroup = new NioEventLoopGroup();
23.         EventLoopGroup workerGroup = new NioEventLoopGroup();
24.         try {
25.             ServerBootstrap b = new ServerBootstrap();
26.             b.group(bossGroup, workerGroup)
```



```

27.         .channel(NioServerSocketChannel.class)
28.         .option(ChannelOption.SO_BACKLOG, 1024)
29.         .childHandler(new ChildChannelHandler());
30.     // 绑定端口, 同步等待成功
31.     ChannelFuture f = b.bind(port).sync();
32.
33.     // 等待服务端监听端口关闭
34.     f.channel().closeFuture().sync();
35. } finally {
36.     // 优雅退出, 释放线程池资源
37.     bossGroup.shutdownGracefully();
38.     workerGroup.shutdownGracefully();
39. }
40. }
41.
42.     private class ChildChannelHandler extends ChannelHandler {
43.     @Override
44.     protected void initChannel(SocketChannel arg0) throws Exception {
45.         arg0.pipeline().addLast(new LineBasedFrameDecoder(1024));
46.
47.         arg0.pipeline().addLast(new StringDecoder());

```

```
48. }
49.     }
50.
51.     /**
52.      * @param args
53.      * @throws Exception
54.      */
55.     public static void main(String[] args) throws Exception {
56.         int port = 8080;
57.         if (args != null && args.length > 0) {
58.             try {
59.                 port = Integer.valueOf(args[0]);
60.             } catch (NumberFormatException e) {
61.                 // 采用默认值
62.             }
63.         }
64.         new TimeServer().bind(port);
65.     }
66. }
```

重点看45~47行，在原来的TimeServerHandler之前新增了两个解码器：第一个是LineBasedFrameDecoder，第二个是StringDecoder。这两个类的功能后续会进行介绍，下面继续看TimeServerHandler的代码修改。

代码清单4-4 Netty时间服务器服务端TimeServerHandler

```
12. public class TimeServerHandler extends ChannelHandlerAdapti
13.
14.     private int counter;
15.
16.     @Override
17.     public void channelRead(ChannelHandlerContext ctx, Obj
18.     throws Exception {
19. String body = (String) msg;

20. System.out.println("The time server receive order : " + b

21.     + " ; the counter is : " + ++counter);

22. String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(l
23.     System.currentTimeMillis()).toString() : "BAD ORDER";
24. currentTime = currentTime + System.getProperty("line.separ
25. ByteBuf resp = Unpooled.copiedBuffer(currentTime.getBytes(
26. ctx.writeAndFlush(resp);
```

```
27.     }
28.
29.     @Override
30.     public void exceptionCaught(ChannelHandlerContext ctx,
31. ctx.close());
32.     }
33. }
```

直接看19~21行，可以发现接收到的msg就是删除回车换行符后的请求消息，不需要额外考虑处理读半包问题，也不需要请求消息进行编码，代码非常简洁。读者可能会质疑这样是否可行，不着急，我们先继续看看客户端的类似改造，然后运行程序看执行结果，最后再揭开其中的奥秘。

4.3.2 支持TCP粘包的TimeClient

支持TCP粘包的客户端修改起来也非常简单，代码如下。

代码清单4-5 Netty时间服务器客户端TimeClient

```
18. public class TimeClient {
19.
20.     public void connect(int port, String host) throws Exception {
21. // 配置客户端NIO线程组
22. EventLoopGroup group = new NioEventLoopGroup();
23. try {
24.     Bootstrap b = new Bootstrap();
```

```
25.         b.group(group).channel(NioSocketChannel.class)
26.             .option(ChannelOption.TCP_NODELAY, true)
27.             .handler(new ChannelInitializer<SocketChannel>() {
28.                 @Override
29.                 public void initChannel(SocketChannel ch)
30.                     throws Exception {
31.                     ch.pipeline().addLast(

32.                         new LineBasedFrameDecoder(1024));

33.                     ch.pipeline().addLast(new StringDecoder());

34.                     ch.pipeline().addLast(new TimeClientHandler(),

35.                 }
36.             });
37.
38.         // 发起异步连接操作
39.         ChannelFuture f = b.connect(host, port).sync();
```

```
40.
41.     // 等待客户端链路关闭
42.     f.channel().closeFuture().sync();
43. } finally {
44.     // 优雅退出，释放NIO线程组
45.     group.shutdownGracefully();
46. }
47. }
48.
49. /**
50.  * @param args
51.  * @throws Exception
52.  */
53. public static void main(String[] args) throws Exception {
54.     int port = 8080;
55.     if (args != null && args.length > 0) {
56.         try {
57.             port = Integer.valueOf(args[0]);
58.         } catch (NumberFormatException e) {
59.             // 采用默认值
60.         }
61.     }
62.     new TimeClient().connect(port, "127.0.0.1");
63. }
64. }
```

31~34行与服务端类似，直接在TimeClientHandler之前新增LineBasedFrameDecoder和StringDecoder解码器，下面我们继续看TimeClientHandler的代码修改。

代码清单4-6 Netty时间服务器客户端TimeClientHandler

```
14. public class TimeClientHandler extends ChannelHandlerAdapti
15.
16.     private static final Logger logger = Logger
17.         .getLogger(TimeClientHandler.class.getName());
18.
19.     private int counter;
20.
21.     private byte[] req;
22.
23.     /**
24.      * Creates a client-side handler.
25.      */
26.     public TimeClientHandler() {
27. req = ("QUERY TIME ORDER" + System.getProperty("line.separ
28.         .getBytes());
29.     }
30.
31.     @Override
32.     public void channelActive(ChannelHandlerContext ctx) {
33. ByteBuf message = null;
34. for (int i = 0; i < 100; i++) {
```

```

35.     message = Unpooled.buffer(req.length);
36.     message.writeBytes(req);
37.     ctx.writeAndFlush(message);
38. }
39. }
40.
41.     @Override
42.     public void channelRead(ChannelHandlerContext ctx, Object msg)
43.         throws Exception {
44.         String body = (String) msg;

45.         System.out.println("Now is : " + body + " ; the counter is

46.         + ++counter);

47.     }
48.
49.     @Override
50.     public void exceptionCaught(ChannelHandlerContext ctx,
51. // 释放资源
52. logger.warning("Unexpected exception from downstream : "

```


[illegible]

[illegible]

[illegible]

The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte
The time server receive order : QUERY TIME ORDER ; the counte

客户端运行结果如下。

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 1
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 2
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 3
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 4
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 5
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 6
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 7
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 8
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 9
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 10
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 11
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 12
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 13
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 14

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 15
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 16
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 17
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 18
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 19
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 20
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 21
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 22
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 23
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 24
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 25
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 26
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 27
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 28
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 29
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 30
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 31
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 32
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 33
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 34
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 35
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 36
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 37
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 38
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 39
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 40
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 41

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 42
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 43
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 44
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 45
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 46
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 47
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 48
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 49
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 50
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 51
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 52
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 53
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 54
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 55
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 56
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 57
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 58
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 59
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 60
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 61
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 62
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 63
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 64
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 65
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 66
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 67
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 68

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 69
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 70
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 71
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 72
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 73
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 74
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 75
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 76
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 77
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 78
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 79
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 80
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 81
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 82
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 83
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 84
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 85
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 86
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 87
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 88
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 89
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 90
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 91
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 92
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 93
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 94
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 95


```
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 96
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 97
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 98
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 99
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 100
```

程序的运行结果完全符合预期，说明通过使用 `LineBasedFrameDecoder` 和 `StringDecoder` 成功解决了 TCP 粘包导致的读半包问题。对于使用者来说，只要将支持半包解码的 `handler` 添加到 `ChannelPipeline` 中即可，不需要写额外的代码，用户使用起来非常简单。

下个小节，我们就对添加 `LineBasedFrameDecoder` 和 `StringDecoder` 之后就能解决 TCP 粘包导致的读半包或者多包问题的原因进行分析。

4.3.4 `LineBasedFrameDecoder` 和 `StringDecoder` 的原理分析

`LineBasedFrameDecoder` 的工作原理是它依次遍历 `ByteBuf` 中的可读字节，判断看是否有“`\n`”或者“`\r\n`”，如果有，就以此位置为结束位置，从可读索引到结束位置区间的字节就组成了一行。它是以换行符为结束标志的解码器，支持携带结束符或者不携带结束符两种解码方式，同时支持配置单行的最大长度。如果连续读取到最大长度后仍然没有发现换行符，就会抛出异常，同时忽略掉之前读到的异常码流。

`StringDecoder` 的功能非常简单，就是将接收到的对象转换成字符串，然后继续调用后面的 `handler`。`LineBasedFrameDecoder` + `StringDecoder` 组合就是按行切换的文本解码器，它被设计用来支持 TCP 的粘包和拆包。

可能读者会提出新的疑问：如果发送的消息不是以换行符结束的该怎么办呢？或者没有回车换行符，靠消息头中的长度字段来分包怎么办？是不是需要自己写半包解码器？答案是否定的，**Netty**提供了多种支持TCP粘包/拆包的解码器，用来满足用户的不同诉求。

第5章我们将学习分隔符解码器，由于它在实际项目中应用非常广泛，所以单独用一章对其用法和原理进行讲解。

4.4 总结

本章首先对TCP的粘包和拆包进行了讲解，给出了解决这个问题的通用做法。然后我们对第3章的时间服务器进行改造和测试，首先验证没有考虑TCP粘包/拆包导致的问题。随后给出了解决方案，即利用LineBasedFrameDecoder+StringDecoder来解决TCP的粘包/拆包问题。

第5章 分隔符和定长解码器的应用

TCP以流的方式进行数据传输，上层的应用协议为了对消息进行区分，往往采用如下4种方式。

（1）消息长度固定，累计读取到长度总和为定长LEN的报文后，就认为读取到了一个完整的消息；将计数器置位，重新开始读取下一个数据报；

（2）将回车换行符作为消息结束符，例如FTP协议，这种方式在文本协议中应用比较广泛；

（3）将特殊的分隔符作为消息的结束标志，回车换行符就是一种特殊的结束分隔符；

（4）通过在消息头中定义长度字段来标识消息的总长度。

Netty对上面四种应用做了统一的抽象，提供了4种解码器来解决对应的问题，使用起来非常方便。有了这些解码器，用户不需要自己对读取的报文进行人工解码，也不需要考虑TCP的粘包和拆包。

第4章我们介绍了如何利用LineBasedFrameDecoder解决TCP的粘包问题，本章我们继续学习另外两种实用的解码器——DelimiterBasedFrameDecoder和FixedLengthFrameDecoder，前者可以自动完成以分隔符做结束标志的消息的解码，后者可以自动完成对定长消息的解码，它们都能解决TCP粘包/拆包导致的读半包问题。

本章主要包括：

- DelimiterBasedFrameDecoder服务端开发
- DelimiterBasedFrameDecoder客户端开发
- 运行DelimiterBasedFrameDecoder服务端和客户端
- FixedLengthFrameDecoder服务端开发
- 通过telnet命令行调试FixedLengthFrameDecoder服务端

5.1 DelimiterBasedFrameDecoder应用开发

通过对DelimiterBasedFrameDecoder的使用，我们可以自动完成以分隔符作为码流结束标识的消息的解码，下面通过一个演示程序来学习下如何使用DelimiterBased FrameDecoder进行开发。

演示程序以经典的Echo服务为例。EchoServer接收到EchoClient的请求消息后，将其打印出来，然后将原始消息返回给客户端，消息以“\$ _”作为分隔符。

5.1.1 DelimiterBasedFrameDecoder服务端开发

下面我们直接看EchoServer的源代码：

代码清单5-1 EchoServer服务端EchoServer

```
22. public class EchoServer {
23.     public void bind(int port) throws Exception {
24.         // 配置服务端的NIO线程组
25.         EventLoopGroup bossGroup = new NioEventLoopGroup();
26.         EventLoopGroup workerGroup = new NioEventLoopGroup(
27.             try {
28.                 ServerBootstrap b = new ServerBootstrap();
29.                 b.group(bossGroup, workerGroup)
30.                     .channel(NioServerSocketChannel.class)
31.                     .option(ChannelOption.SO_BACKLOG, 100)
32.                     .handler(new LoggingHandler(LogLevel.INFO))
```

```
33.         .childHandler(new ChannelInitializer<Socket
34.         @Override
35.         public void initChannel(SocketChannel ch)
36.             throws Exception {
37.             ByteBuf delimiter = Unpooled.copiedBuff

38.             .getBytes());

39.             ch.pipeline().addLast(

40.             new DelimiterBasedFrameDecoder(1024

41.             delimiter));

42.             ch.pipeline().addLast(new StringDecoder
43.             ch.pipeline().addLast(new EchoServerHan
44.         }
```

```
45.         });
46.
47.         // 绑定端口，同步等待成功
48.         ChannelFuture f = b.bind(port).sync();
49.
50.         // 等待服务端监听端口关闭
51.         f.channel().closeFuture().sync();
52.     } finally {
53.         // 优雅退出，释放线程池资源
54.         bossGroup.shutdownGracefully();
55.         workerGroup.shutdownGracefully();
56.     }
57. }
58.
59. public static void main(String[] args) throws Excep
60. int port = 8080;
61. if (args != null && args.length > 0) {
62.     try {
63.         port = Integer.valueOf(args[0]);
64.     } catch (NumberFormatException e) {
65.         // 采用默认值
66.     }
67. }
68. new EchoServer().bind(port);
69. }
70. }
```

我们重点看37~41行，首先创建分隔符缓冲对象ByteBuf，本例程中使用“\$_”作为分隔符。第40行，创建DelimiterBasedFrameDecoder对象，将其加入到ChannelPipeline中。DelimiterBasedFrameDecoder有多个构造方法，这里我们传递两个参数，第一个1024表示单条消息的最大长度，当达到该长度后仍然没有查找到分隔符，就抛出TooLongFrame Exception异常，防止由于异常码流缺失分隔符导致的内存溢出，这是Netty解码器的可靠性保护；第二个参数就是分隔符缓冲对象。

下面继续看EchoServerHandler的实现。

代码清单5-2 EchoServer服务端EchoServerHandler

```
13.  @Sharable
14.  public class EchoServerHandler extends ChannelHandlerAd
15.
16.      int counter = 0;
17.
18.      @Override
19.      public void channelRead(ChannelHandlerContext ctx,
20.          throws Exception {
21.          String body = (String) msg;

22.          System.out.println("This is " + ++counter + " times
```

```
23.         + body + "]"");
```

```
24.         body += "$_";
```

```
25.         ByteBuf echo = Unpooled.copiedBuffer(body.getBytes(
```

```
26.         ctx.writeAndFlush(echo);
```

```
27.     }
```

```
28.
```

```
29.     @Override
```

```
30.     public void exceptionCaught(ChannelHandlerContext c
```

```
31.     cause.printStackTrace();
```

```
32.     ctx.close(); // 发生异常，关闭链路
```

```
33. }
```

```
34. }
```

第21~23行直接将接收的消息打印出来，由于DelimiterBasedFrameDecoder自动对请求消息进行了解码，后续的ChannelHandler接收到的msg对象就是个完整的消息包；第二个ChannelHandler是StringDecoder，它将ByteBuf解码成字符串对象；第三个EchoServerHandler接收到的msg消息就是解码后的字符串对象。

由于我们设置DelimiterBasedFrameDecoder过滤掉了分隔符，所以，返回给客户端时需要在请求消息尾部拼接分隔符“\$ _”，最后创建ByteBuf，将原始消息重新返回给客户端。

下面我们继续看下客户端的实现。

5.1.2 DelimiterBasedFrameDecoder客户端开发

首先看下EchoClient的实现。

代码清单5-3 EchoClient客户端EchoClient

```
20. public class EchoClient {
21.
22.     public void connect(int port, String host) throws E
23.         // 配置客户端NIO线程组
24.         EventLoopGroup group = new NioEventLoopGroup();
25.     try {
26.         Bootstrap b = new Bootstrap();
27.         b.group(group).channel(NioSocketChannel.class)
28.             .option(ChannelOption.TCP_NODELAY, true)
29.             .handler(new ChannelInitializer<SocketChann
```

```
30.         @Override
31.         public void initChannel(SocketChannel ch)
32.             throws Exception {
33.             ByteBuf delimiter = Unpooled.copiedBuff

34.                 .getBytes());

35.             ch.pipeline().addLast(

36.                 new DelimiterBasedFrameDecoder(1024

37.                     delimiter));

38.             ch.pipeline().addLast(new StringDecoder
```

```

39.                ch.pipeline().addLast(new EchoClientHan

40.                }

41.                });

42.

43.                // 发起异步连接操作

44.                ChannelFuture f = b.connect(host, port).sync();

45.

46.                // 等待客户端链路关闭

47.                f.channel().closeFuture().sync();

48.        } finally {

49.            // 优雅退出，释放NIO线程组

50.            group.shutdownGracefully();

51.        }

52.    }

53.

54.    /**

55.     * @param args

56.     * @throws Exception

57.     */

58.    public static void main(String[] args) throws Excep

59.    int port = 8080;

60.    if (args != null && args.length > 0) {

61.        try {

62.            port = Integer.valueOf(args[0]);

```

```
63.         } catch (NumberFormatException e) {
64.             // 采用默认值
65.         }
66.     }
67.     new EchoClient().connect(port, "127.0.0.1");
68. }
69. }
```

与服务端类似，分别将DelimiterBasedFrameDecoder和StringDecoder添加到客户端ChannelPipeline中，最后添加客户端I/O事件处理类EchoClientHandler，下面继续看EchoClientHandler的实现。

代码清单5-4 EchoClient客户端EchoClientHandler

```
11. public class EchoClientHandler extends ChannelHandlerAd
12.
13.     private int counter;
14.
15.     static final String ECHO_REQ = "Hi, Lilinfeng. Welc
16.
17.     /**
18.      * Creates a client-side handler.
19.      */
20.     public EchoClientHandler() {
21.     }
22.
```

```
23.      @Override
24.      public void channelActive(ChannelHandlerContext ctx
25.      for (int i = 0; i < 10; i++) {          ctx.writeAn

26.      }

27.      }

28.
29.      @Override
30.      public void channelRead(ChannelHandlerContext ctx,
31.      throws Exception {
32.      System.out.println("This is " + ++counter + " times

33.      + msg + "]);

34.      }
```

```
35.  
36.      @Override  
37.      public void channelReadComplete(ChannelHandlerContext  
38.      ctx.flush();  
39.      }  
40.  
41.      @Override  
42.      public void exceptionCaught(ChannelHandlerContext c  
43.      cause.printStackTrace();  
44.      ctx.close();  
45.      }  
46.  }
```

第25~26行在TCP链路建立成功之后循环发送请求消息给服务端，第32~33行打印接收到的服务端应答消息同时进行计数。

下个小节，运行上面开发的服务端和客户端，看看运行结果是否正确。

5.1.3 运行**DelimiterBasedFrameDecoder**服务端和客户端

服务端运行结果如下。

```
This is 1 times receive client : [Hi, Lilinfeng. Welcome to N  
This is 2 times receive client : [Hi, Lilinfeng. Welcome to N  
This is 3 times receive client : [Hi, Lilinfeng. Welcome to N  
This is 4 times receive client : [Hi, Lilinfeng. Welcome to N
```



```
This is 5 times receive client : [Hi, Lilinfeng. Welcome to N
This is 6 times receive client : [Hi, Lilinfeng. Welcome to N
This is 7 times receive client : [Hi, Lilinfeng. Welcome to N
This is 8 times receive client : [Hi, Lilinfeng. Welcome to N
This is 9 times receive client : [Hi, Lilinfeng. Welcome to N
This is 10 times receive client : [Hi, Lilinfeng. Welcome to
```

客户端运行结果如下。

```
This is 1 times receive server : [Hi, Lilinfeng. Welcome to N
This is 2 times receive server : [Hi, Lilinfeng. Welcome to N
This is 3 times receive server : [Hi, Lilinfeng. Welcome to N
This is 4 times receive server : [Hi, Lilinfeng. Welcome to N
This is 5 times receive server : [Hi, Lilinfeng. Welcome to N
This is 6 times receive server : [Hi, Lilinfeng. Welcome to N
This is 7 times receive server : [Hi, Lilinfeng. Welcome to N
This is 8 times receive server : [Hi, Lilinfeng. Welcome to N
This is 9 times receive server : [Hi, Lilinfeng. Welcome to N
This is 10 times receive server : [Hi, Lilinfeng. Welcome to
```

服务端成功接收到了客户端发送的10条“Hi, Lilinfeng. Welcome to Netty.”请求消息，客户端成功接收到了服务端返回的10条“Hi, Lilinfeng. Welcome to Netty.”应答消息。测试结果表明使用DelimiterBasedFrameDecoder可以自动对采用分隔符做码流结束标识的消息进行解码。

本例程运行10次的原因是模拟TCP粘包/拆包，在笔者的机器上，连续发送10条Echo请求消息会发生粘包，如果没有DelimiterBasedFrameDecoder解码器的处理，服务端和客户端程序都将运行失败。下面我们将服务端的DelimiterBasedFrameDecoder注释掉，最终代码如图5-1所示。

图5-1 删除掉DelimiterBasedFrameDecoder后的服务端代码

服务端运行结果如下。

```
This is 1 times receive client : [Hi, Lilinfeng. Welcome to N
```

由于没有分隔符解码器，导致服务端一次读取了客户端发送的所有消息，这就是典型的没有考虑TCP粘包导致的问题。

5.2 FixedLengthFrameDecoder应用开发

FixedLengthFrameDecoder是固定长度解码器，它能够按照指定的长度对消息进行自动解码，开发者不需要考虑TCP的粘包/拆包问题，非常实用。下面我们通过一个应用实例对其用法进行讲解。

5.2.1 FixedLengthFrameDecoder服务端开发

在服务端的ChannelPipeline中新增FixedLengthFrameDecoder，长度设置为20，然后再依次增加字符串解码器和EchoServerHandler，代码如下。

代码清单5-5 EchoServer服务端 EchoServer

```
20. public class EchoServer {
21.     public void bind(int port) throws Exception {
22.         // 配置服务端的NIO线程组
23.         EventLoopGroup bossGroup = new NioEventLoopGroup();
24.         EventLoopGroup workerGroup = new NioEventLoopGroup(
25.             try {
26.                 ServerBootstrap b = new ServerBootstrap();
27.                 b.group(bossGroup, workerGroup)
28.                     .channel(NioServerSocketChannel.class)
29.                     .option(ChannelOption.SO_BACKLOG, 100)
30.                     .handler(new LoggingHandler(LogLevel.INFO))
31.                     .childHandler(new ChannelInitializer<Socket
32. @Override
```

```
33.         public void initChannel(SocketChannel ch)
34.             throws Exception {
35.             ch.pipeline().addLast(

36.                 new FixedLengthFrameDecoder(20));

37.             ch.pipeline().addLast(new StringDecoder
38.             ch.pipeline().addLast(new EchoServerHan
39.         }
40.     });
41.
42.     // 绑定端口，同步等待成功
43.     ChannelFuture f = b.bind(port).sync();
44.
45.     // 等待服务端监听端口关闭
46.     f.channel().closeFuture().sync();
47. } finally {
48.     // 优雅退出，释放线程池资源
49.     bossGroup.shutdownGracefully();
50.     workerGroup.shutdownGracefully();
51. }
52. }
53.
```

```
54.         public static void main(String[] args) throws Excep
55.         int port = 8080;
56.         if (args != null && args.length > 0) {
57.             try {
58.                 port = Integer.valueOf(args[0]);
59.             } catch (NumberFormatException e) {
60.                 // 采用默认值
61.             }
62.         }
63.         new EchoServer().bind(port);
64.     }
65. }
```

EchoServerHandler的功能比较简单，直接将读取到的消息打印出来，代码如下。

代码清单5-6 EchoServer服务端 EchoServerHandler

```
11.     @Sharable
12.     public class EchoServerHandler extends ChannelHandlerAd
13.
14.         @Override
15.         public void channelRead(ChannelHandlerContext ctx,
16.             throws Exception {
17.             System.out.println("Receive client : [" + msg + "]"
18.         }
```

```
19.  
20.      @Override  
21.      public void exceptionCaught(ChannelHandlerContext c  
22.      cause.printStackTrace();  
23.      ctx.close()); // 发生异常，关闭链路  
24.      }  
25.  }
```

利用FixedLengthFrameDecoder解码器，无论一次接收到多少数据报，它都会按照构造函数中设置的固定长度进行解码，如果是半包消息，FixedLengthFrameDecoder会缓存半包消息并等待下个包到达后进行拼包，直到读取到一个完整的包。

下面的章节我们通过telnet命令行来测试EchoServer 服务端，看它能否按照预期进行工作。

5.2.2 利用telnet命令行测试EchoServer服务端

由于客户端代码比较简单，所以这次我们通过telnet命令行对服务端进行测试。

测试场景：在Windows操作系统上打开CMD命令行窗口，通过telnet命令行连接服务端，在控制台输入如下内容。

```
Lilinfeng welcome to Netty at Nanjing
```

然后看服务端打印的内容，预期输出的请求消息为“Lilinfeng

welcome to”。

下面我们就具体看下详细的测试步骤。

(1) 在【运行】菜单中输入cmd命令，打开命令行窗口，如图5-2所示。

图5-2 通过cmd命令打开CMD窗口

(2) 在命令行中输入“telnet localhost 8080”，通过telnet连接服务端，如图5-3所示。

图5-3 通过telnet命令连接服务端

(3) 通过set localecho命令打开本地回显功能，输入命令行内容，如图5-4所示。

图5-4 输入Lilinfeng welcome to Netty at Nanjing

(4) EchoServer服务端运行结果如图5-5所示。

图5-5 服务端运行结果

根据图5-5所示内容，服务端运行结果完全符合预期，FixedLengthFrameDecoder解码器按照20个字节长度对请求消息进行截取，输出结果为“Lilinfeng welcome to”。

5.3 总结

本章我们学习了两个非常实用的解码器：
`DelimiterBasedFrameDecoder`和`FixedLengthFrameDecoder`。

`DelimiterBasedFrameDecoder`用于对使用分隔符结尾的消息进行自动解码，`FixedLengthFrameDecoder`用于对固定长度的消息进行自动解码。有了上述两种解码器，再结合其他的解码器，如字符串解码器等，可以轻松地完成对很多消息的自动解码，而且不再需要考虑TCP粘包/拆包导致的读半包问题，极大地提升了开发效率。

应用`DelimiterBasedFrameDecoder`和`FixedLengthFrameDecoder`进行开发非常简单，在绝大多数情况下，只要将`DelimiterBasedFrameDecoder`或`FixedLengthFrameDecoder`添加到对应`ChannelPipeline`的起始位即可。

熟悉了Netty的NIO基础应用开发之后，从第三部分开始，我们继续学习编解码技术。在了解编解码基础知识之后，继续学习Netty内置的编解码框架的使用，例如Java序列化、二进制编解码、谷歌的protobuf和JBoss的Marshalling序列化框架。

中级篇 **Netty**编解码开发指南

第6章 编解码技术

第7章 Java序列化

第8章 Google Protobuf编解码

第9章 JBoss Marshalling编解码

第6章 编解码技术

基于Java提供的对象输入/输出流ObjectInputStream和ObjectOutputStream，可以直接把Java对象作为可存储的字节数组写入文件，也可以传输到网络上。对程序员来说，基于JDK默认的序列化机制可以避免操作底层的字节数组，从而提升开发效率。

Java序列化的目的主要有两个：

- 网络传输
- 对象持久化

由于本书主要介绍基于Netty的NIO网络开发，所以我们重点关注网络传输。当进行远程跨进程服务调用时，需要把被传输的Java对象编码为字节数组或者ByteBuffer对象。而当远程服务读取到ByteBuffer对象或者字节数组时，需要将其解码为发送时的Java对象。这被称为Java对象编解码技术。

Java序列化仅仅是Java编解码技术的一种，由于它的种种缺陷，衍生出了多种编解码技术和框架，后续的章节我们会结合Netty介绍几种业界主流的编解码技术和框架，看看如何在Netty中应用这些编解码框架实现消息的高效序列化。

本章主要包括：

- Java序列化的缺点
- 业界流行的几种编解码框架介绍

6.1 Java序列化的缺点

Java序列化从JDK 1.1版本就已经提供，它不需要添加额外的类库，只需实现`java.io.Serializable`并生成序列ID即可，因此，它从诞生之初就得到了广泛的应用。

但是在远程服务调用（RPC）时，很少直接使用Java序列化进行消息的编解码和传输，这又是什么原因呢？下面通过分析Java序列化的缺点来找出答案。

6.1.1 无法跨语言

无法跨语言，是Java序列化最致命的问题。对于跨进程的服务调用，服务提供者可能会使用C++或者其他语言开发，当我们需要和异构语言进程交互时，Java序列化就难以胜任。

由于Java序列化技术是Java语言内部的私有协议，其他语言并不支持，对于用户来说它完全是黑盒。对于Java序列化后的字节数组，别的语言无法进行反序列化，这就严重阻碍了它的应用。

事实上，目前几乎所有流行的Java RPC通信框架，都没有使用Java序列化作为编解码框架，原因就在于它无法跨语言，而这些RPC框架往往需要支持跨语言调用。

6.1.2 序列化后的码流太大

下面我们通过一个实例看下Java序列化后的字节数组大小。

代码清单6-1 Java序列化代码 POJO对象类UserInfo

```
10. public class UserInfo implements Serializable {
11.
12.     /**
13.      * 默认的序列号
14.      */
15.     private static final long serialVersionUID = 1L;
16.
17.     private String userName;
18.
19.     private int userID;
20.
21.     public UserInfo buildUserName(String userName) {
22.         this.userName = userName;
23.         return this;
24.     }
25.
26.     public UserInfo buildUserID(int userID) {
27.         this.userID = userID;
28.         return this;
29.     }
30.
31.     /**
32.      * @return the userName
33.      */
34.     public final String getUserName() {
35.         return userName;
```

```
36.     }
37.
38.     /**
39.      * @param userName
40.      *         the userName to set
41.      */
42.     public final void setUsername(String userName) {
43.         this.userName = userName;
44.     }
45.
46.     /**
47.      * @return the userID
48.      */
49.     public final int getUserID() {
50.         return userID;
51.     }
52.
53.     /**
54.      * @param userID
55.      *         the userID to set
56.      */
57.     public final void setUserID(int userID) {
58.         this.userID = userID;
59.     }
60.
61.     public byte[] codeC() {
62.         ByteBuffer buffer = ByteBuffer.allocate(1024);
```

```
63.      byte[] value = this.userName.getBytes();
64.      buffer.putInt(value.length);
65.      buffer.put(value);
66.      buffer.putInt(this.userID);
67.      buffer.flip();
68.      value = null;
69.      byte[] result = new byte[buffer.remaining()];
70.      buffer.get(result);
71.      return result;
72.  }
73. }
```

UserInfo对象是个普通的POJO对象，它实现了java.io.Serializable接口，并且生成了一个默认的序列号serialVersionUID = 1L。这说明UserInfo对象可以通过JDK默认的序列化机制进行序列化和反序列化。

第61~72行使用基于ByteBuffer的通用二进制编解码技术对UserInfo对象进行编码，编码结果仍然是byte数组，可以与传统的JDK序列化后的码流大小进行对比。

下面写一个测试程序，先调用两种编码接口对POJO对象编码，然后分别打印两者编码后的码流大小进行对比。

代码清单6-2 Java序列化代码 编码测试类TestUserInfo

```
11. public class TestUserInfo {
12.
```

```
13.      /**
14.      * @param args
15.      * @throws IOException
16.      */
17.      public static void main(String[] args) throws IOExc
18.      {
19.          UserInfo info = new UserInfo();
20.          info.buildUserID(100).buildUserName("Welcome to Net
21.          ByteArrayOutputStream bos = new ByteArrayOutputStre
22.          ObjectOutputStream os = new ObjectOutputStream(bos)
23.          os.writeObject(info);
24.          os.flush();
25.          os.close();
26.          byte[] b = bos.toByteArray();
27.          System.out.println("The jdk serializable length is
28.          bos.close();
29.          System.out.println("-----
30.          System.out.println("The byte array serializable len
31.          + info.codeC().length);
32.      }
33.  }
```

测试结果如图6-1所示。

图6-1 JDK序列化机制和通用二进制编码测试结果

测试结果令人震惊，采用JDK序列化机制编码后的二进制数组大小

竟然是二进制编码的5.29倍。

我们评判一个编解码框架的优劣时，往往会考虑以下几个因素。

- 是否支持跨语言，支持的语言种类是否丰富；
- 编码后的码流大小；
- 编解码的性能；
- 类库是否小巧，API使用是否方便；
- 使用者需要手工开发的工作量和难度。

在同等情况下，编码后的字节数组越大，存储的时候就越占空间，存储的硬件成本就越高，并且在网络传输时更占带宽，导致系统的吞吐量降低。Java序列化后的码流偏大也一直被业界所诟病，导致它的应用范围受到了很大限制。

6.1.3 序列化性能太低

下面我们从序列化的性能角度看下JDK的表现如何。将之前的例程代码稍做修改，改造成性能测试版本，如图6-2所示。

图6-2 UserInfo性能测试版本修改

对UserInfo进行改造，新增上图所示的方法，再创建一个性能测试版本的UserInfo测试程序，代码如下。

代码清单6-3 Java序列化代码 编码性能测试类
PerformTestUserInfo

```
12. public class PerformTestUserInfo {
```



```
13.
14.     /**
15.      * @param args
16.      * @throws IOException
17.      */
18.     public static void main(String[] args) throws IOExc
19.     {
20.         UserInfo info = new UserInfo();
21.         info.buildUserID(100).buildUserName("Welcome to Net
22.         int loop = 1000000;
23.         ByteArrayOutputStream bos = null;
24.         ObjectOutputStream os = null;
25.         long startTime = System.currentTimeMillis();
26.         for (int i = 0; i < loop; i++) {
27.             bos = new ByteArrayOutputStream();
28.             os = new ObjectOutputStream(bos);
29.             os.writeObject(info);
30.             os.flush();
31.             os.close();
32.             byte[] b = bos.toByteArray();
33.             bos.close();
34.         }
35.         long endTime = System.currentTimeMillis();
36.         System.out.println("The jdk serializable cost time
37.             + (endTime - startTime) + " ms");
38.
39.         System.out.println("-----");
```

```
40.         ByteBuffer buffer = ByteBuffer.allocate(1024);
41.         startTime = System.currentTimeMillis();
42.         for (int i = 0; i < loop; i++) {
43.             byte[] b = info.codec(buffer);
44.         }
45.         endTime = System.currentTimeMillis();
46.         System.out.println("The byte array serializable cos
47.             + (endTime - startTime) + " ms");
48.     }
49. }
```

对Java序列化和二进制编码分别进行性能测试，编码100万次，然后统计耗费的总时间，测试结果如图6-3所示。

图6-3 UserInfo编码性能测试结果

这个结果也非常令人惊讶：Java序列化的性能只有二进制编码的6.17%左右，可见Java原生序列化的性能实在太差。

下面我们结合编码速度，综合对比一下Java序列化和二进制编码的性能差异，如图6-4所示。

图6-4 序列化性能对比图

从图6-4可以看出，无论是序列化后的码流大小，还是序列化的性能，JDK默认的序列化机制表现得都很差。因此，我们通常不会选择Java序列化作为远程跨节点调用的编解码框架。

但是不使用JDK提供的默认序列化框架，自己开发编解码框架又是

个非常复杂的工作，怎么办呢？不用着急，业界有很多优秀的编解码框架，它们在克服了JDK默认序列化框架缺点的基础上，还增加了很多亮点，下面让我们继续了解并学习业界流行的几款编解码框架。

6.2 业界主流的编解码框架

由于Java的编解码框架五花八门，穷举学习显然不是一个好的策略，本节挑选了一些业界主流的编解码框架和编解码技术进行介绍，希望读者在了解这些框架特性的基础上，做出合理的选择。

6.2.1 Google的Protobuf介绍

Protobuf全称Google Protocol Buffers，它由谷歌开源而来，在谷歌内部久经考验。它将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的POJO对象和Protobuf相关的方法和属性。

它的特点如下。

- 结构化数据存储格式（XML，JSON等）；
- 高效的编解码性能；
- 语言无关、平台无关、扩展性好；
- 官方支持Java、C++和Python三种语言。

首先我们来看下为什么不使用XML，尽管XML的可读性和可扩展性非常好，也非常适合描述数据结构，但是XML解析的时间开销和XML为了可读性而牺牲的空间开销都非常大，因此不适合做高性能的通信协议。Protobuf使用二进制编码，在空间和性能上具有更大的优势。

Protobuf另一个比较吸引人的地方就是它的数据描述文件和代码生成机制，利用数据描述文件对数据结构进行说明的优点如下。

- 文本化的数据结构描述语言，可以实现语言和平台无关，特别适合异构系统间的集成；
- 通过标识字段的顺序，可以实现协议的前向兼容；
- 自动代码生成，不需要手工编写同样数据结构的C++和Java版本；
- 方便后续的管理和维护。相比于代码，结构化的文档更容易管理和维护。

下面我们看下Protobuf编解码和其他几种序列化框架的性能对比数据，如图6-5、图6-6所示。

图6-5 Protobuf编解码和其他几种序列化框架的响应时间对比

图6-6 Protobuf和其他几种序列化框架的字节数对比

从图6-5和图6-6两幅对比图可以发现，Protobuf的编解码性能远远高于其他几种序列化框架的序列化和反序列化，这也是很多RPC框架选用Protobuf做编解码框架的原因。

6.2.2 Facebook的Thrift介绍

Thrift源于Facebook，在2007年Facebook将Thrift作为一个开源项目提交给Apache基金会。对于当时的Facebook来说，创造Thrift是为了解决Facebook各系统间大数据量的传输通信以及系统之间语言环境不同需要跨平台的特性，因此Thrift可以支持多种程序语言，如C++、C#、Cocoa、Erlang、Haskell、Java、Ocaml、Perl、PHP、Python、Ruby和Smalltalk。

在多种不同的语言之间通信，Thrift可以作为高性能的通信中间件使用，它支持数据（对象）序列化和多种类型的RPC服务。Thrift适用

于静态的数据交换，需要先确定好它的数据结构，当数据结构发生变化时，必须重新编辑IDL文件，生成代码和编译，这一点跟其他IDL工具相比可以视为是Thrift的弱项。Thrift适用于搭建大型数据交换及存储的通用工具，对于大型系统中的内部数据传输，相对于JSON和XML在性能和传输大小上都有明显的优势。

Thrift主要由5部分组成。

（1）语言系统以及IDL编译器：负责由用户给定的IDL文件生成相应语言的接口代码；

（2）TProtocol：RPC的协议层，可以选择多种不同的对象序列化方式，如JSON和Binary；

（3）TTransport：RPC的传输层，同样可以选择不同的传输层实现，如socket、NIO、MemoryBuffer等；

（4）TProcessor：作为协议层和用户提供的服务实现之间的纽带，负责调用服务实现的接口；

（5）TServer：聚合TProtocol、TTransport和TProcessor等对象。

我们重点关注的是编解码框架，与之对应的就是TProtocol。由于Thrift的RPC服务调用和编解码框架绑定在一起，所以，通常我们使用Thrift的时候会采取RPC框架的方式。但是，它的TProtocol编解码框架还是可以以类库的方式独立使用的。

与Protobuf比较类似的是，Thrift通过IDL描述接口和数据结构定义，它支持8种Java基本类型、Map、Set和List，支持可选和必选定义，功能非常强大。因为可以定义数据结构中字段的顺序，所以它也可以支

持协议的前向兼容。

Thrift支持三种比较典型的编解码方式。

- 通用的二进制编解码；
- 压缩二进制编解码；
- 优化的可选字段压缩编解码。

由于支持二进制压缩编解码，Thrift的编解码性能表现也相当优异，远远超过Java序列化和RMI等，图6-7展示了同等测试条件下的编解码耗时信息。

图6-7 Thrift性能测试对比图

6.2.3 JBoss Marshalling介绍

JBoss Marshalling是一个Java对象的序列化API包，修正了JDK自带的序列化包的很多问题，但又保持跟java.io.Serializable接口的兼容；同时增加了一些可调的参数和附加的特性，并且这些参数和特性可通过工厂类进行配置。

相比于传统的Java序列化机制，它的优点如下：

- 可插拔的类解析器，提供更加便捷的类加载定制策略，通过一个接口即可实现定制；
- 可插拔的对象替换技术，不需要通过继承的方式；
- 可插拔的预定义类缓存表，可以减小序列化的字节数组长度，提升常用类型的对象序列化性能；
- 无须实现java.io.Serializable接口，即可实现Java序列化；

- 通过缓存技术提升对象的序列化性能。

相比于前面介绍的两种编解码框架，JBoss Marshalling更多是在JBoss内部使用，应用范围有限。

JBoss Marshalling的使用非常简单，后续在介绍Netty的Marshalling解码器时会给出例程。

6.3 总结

本章首先对Java的序列化技术进行了介绍，对Java序列化的缺点进行了总结说明，在此基础上引出了几款业界主流的编解码框架。由于编解码框架种类繁多，无法一一枚举，所以重点介绍了当前最流行的几种编解码框架。后续在第7章我们会对这些编解码框架的使用进行说明，并给出具体的示例，同时，讲解如何在Netty中应用这些编解码框架。

第7章 **Java**序列化

相信大多数Java程序员接触到的第一种序列化或者编解码技术就是Java的默认序列化，只需要序列化的POJO对象实现`java.io.Serializable`接口，根据实际情况生成序列ID，这个类就能够通过`java.io.ObjectInput`和`java.io.ObjectOutput`序列化和反序列化。

不需要考虑跨语言调用，对序列化的性能也没有苛刻的要求时，Java默认的序列化机制是最明智的选择之一。正因为此，虽然Java序列化机制存在着一些弊病，却依然得到了广泛的应用。

本章主要包括：

- Netty Java序列化服务端开发
- Netty Java序列化客户端开发
- 运行Java序列化应用例程

7.1 Netty Java序列化服务端开发

服务端开发的场景如下：Netty服务端接收到客户端的用户订购请求消息，消息定义如表7-1所示。

表7-1 SubscribeReq消息定义

服务端接收到请求消息，对用户名进行合法性校验。如果合法，则构造订购成功的应答消息返回给客户端。订购应答消息的定义如表7-2所示。

表7-2 SubscribeResp消息定义

本例程中我们将使用Netty的ObjectEncoder和ObjectDecoder对订购请求和应答消息进行序列化。

服务端开发例程

使用Netty对POJO对象进行序列化的开发步骤如下。

（1）在服务端ChannelPipeline中新增解码器
`io.netty.handler.codec.serialization.Object Decoder;`

（2）在服务端ChannelPipeline中新增编码器
`io.netty.handler.codec.serialization.Object Encoder;`

（3）需要进行Java序列化的POJO对象必须实现`java.io.Serializable`接口。

下面我们通过产品订购例程来学习如何在Netty中对POJO对象进行Java序列化。

代码清单7-1 Netty Java序列化 订购请求POJO类定义

```
9.      public class SubscribeReq implements Serializable {
10.
11.          /**
12.           * 默认的序列号ID
13.           */
14.          private static final long serialVersionUID = 1L;
15.
16.          private int subReqID;
17.
18.          private String userName;
19.
20.          private String productName;
21.
22.          private String phoneNumber;
23.
24.          private String address;
.....//get和set方法
100.
101.         /*
102.          * (non-Javadoc)
103.          *
104.          * @see java.lang.Object#toString()
```

```
105.         */
106.         @Override
107.         public String toString() {
108.             return "SubscribeReq [subReqID=" + subReqID + ",
109.                 + ", productName=" + productName + ", phoneNu
110.                 + phoneNumber + ", address=" + address + "];
111.         }
112.     }
```

SubscribeReq是个普通的JOJO对象，需要强调的有两点。

- (1) 第9行实现Serializable接口；
- (2) 第14行自动生成默认的序列化ID。

下面继续看订购应答POJO类。

代码清单7-2 Netty Java序列化 订购应答POJO类定义

```
9.     public class SubscribeResp implements Serializable {
10.
11.         /**
12.          * 默认序列ID
13.          */
14.         private static final long serialVersionUID = 1L;
15.
16.         private int subReqID;
17.
```

```
18.         private int respCode;
19.
20.         private String desc;
//get和set方法.....
66.
67.         /*
68.          * (non-Javadoc)
69.          *
70.          * @see java.lang.Object#toString()
71.          */
72.         @Override
73.         public String toString() {
74.             return "SubscribeResp [subReqID=" + subReqID + ", r
75.                 + ", desc=" + desc + " ]";
76.         }
77.     }
```

应答消息非常简单，我们继续看订购服务主函数定义。

代码清单7-3 Netty Java序列化 订购服务端主函数SubReqServer

```
21.     public class SubReqServer {
22.         public void bind(int port) throws Exception {
23.             // 配置服务端的NIO线程组
24.             EventLoopGroup bossGroup = new NioEventLoopGroup();
25.             EventLoopGroup workerGroup = new NioEventLoopGroup(
```

```
26.         try {
27.             ServerBootstrap b = new ServerBootstrap();
28.             b.group(bossGroup, workerGroup)
29.                 .channel(NioServerSocketChannel.class)
30.                 .option(ChannelOption.SO_BACKLOG, 100)
31.                 .handler(new LoggingHandler(LogLevel.INFO))
32.                 .childHandler(new ChannelInitializer<Socket
33. @Override
34.             public void initChannel(SocketChannel ch) {
35.                 ch.pipeline()

36.                     .addLast(

37.                         new ObjectDecoder(

38.                             1024 * 1024,

39.                             ClassResolvers
```

```
40.                                     .weakCachingConcurrentR

41.                                     .getClass()

42.                                     .getClassLoader()))

43.         ch.pipeline().addLast(new ObjectEncoder

44.         ch.pipeline().addLast(new SubReqServerH

45.         }

46.         });

47.

48.         // 绑定端口，同步等待成功

49.         ChannelFuture f = b.bind(port).sync();
```



```
50.
51.         // 等待服务端监听端口关闭
52.         f.channel().closeFuture().sync();
53.     } finally {
54.         // 优雅退出，释放线程池资源
55.         bossGroup.shutdownGracefully();
56.         workerGroup.shutdownGracefully();
57.     }
58. }
59.
60.     public static void main(String[] args) throws Excep
61.     int port = 8080;
62.     if (args != null && args.length > 0) {
63.         try {
64.             port = Integer.valueOf(args[0]);
65.         } catch (NumberFormatException e) {
66.             // 采用默认值
67.         }
68.     }
69.     new SubReqServer().bind(port);
70. }
71. }
```

从35行开始进行分析。首先创建了一个新的ObjectDecoder，它负责对实现Serializable的POJO对象进行解码，它有多个构造函数，支持不同的ClassResolver，在此我们使用weakCachingConcurrentResolver创建线

程安全的WeakReferenceMap对类加载器进行缓存，它支持多线程并发访问，当虚拟机内存不足时，会释放缓存中的内存，防止内存泄漏。为了防止异常码流和解码错位导致的内存溢出，这里将单个对象最大序列化后的字节数组长度设置为1M，作为例程它已经足够使用。

第43行新增了一个ObjectEncoder，它可以在消息发送的时候自动将实现Serializable的POJO对象进行编码，因此用户无须亲自对对象进行手工序列化，只需要关注自己的业务逻辑处理即可，对象序列化和反序列化都由Netty的对象编解码器搞定。

第44行将订购处理handler SubReqServerHandler添加到ChannelPipeline的尾部用于业务逻辑处理，下面我们看下SubReqServerHandler是如何实现的。

代码清单7-4 Netty Java序列化 订购服务处理类
SubReqServerHandler

```
14.  @Sharable
15.  public class SubReqServerHandler extends ChannelHandler
16.
17.      @Override
18.      public void channelRead(ChannelHandlerContext ctx,
19.          throws Exception {
20.          SubscribeReq req = (SubscribeReq) msg;
21.          if ("Lilinfeng".equalsIgnoreCase(req.getUserName()))
22.              System.out.println("Service accept client subsc
23.                  + req.toString() + "]);
```

```
24.         ctx.writeAndFlush(resp(req.getSubReqID()));
25.     }
26. }
27.
28.     private SubscribeResp resp(int subReqID) {
29.         SubscribeResp resp = new SubscribeResp();
30.         resp.setSubReqID(subReqID);
31.         resp.setRespCode(0);
32.         resp.setDesc("Netty book order succeed, 3 days late
33.         return resp;
34.     }
35.
36.     @Override
37.     public void exceptionCaught(ChannelHandlerContext c
38.         cause.printStackTrace();
39.         ctx.close();// 发生异常，关闭链路
40.     }
41. }
```

经过解码器handler `ObjectDecoder`的解码，`SubReqServerHandler`接收到的请求消息已经被自动解码为`SubscribeReq`对象，可以直接使用。第21行对订购者的用户名进行合法性校验，校验通过后打印订购请求消息，构造订购成功应答消息立即发送给客户端。

下面继续进行产品订购客户端的开发。

7.2 Java序列化Netty客户端开发

客户端的设计思路如下。

(1) 创建客户端的时候将Netty对象解码器和编码器添加到ChannelPipeline;

(2) 链路被激活的时候构造订购请求消息发送，为了检验Netty的Java序列化功能是否支持TCP粘包/拆包，客户端一次构造10条订购请求，最后一次性发送给服务端;

(3) 客户端订购处理handler将接收到的订购响应消息打印出来。

下面我们具体看下客户端的代码实现。

客户端开发例程

代码清单7-5 Netty Java序列化 产品订购客户端

```
19. public class SubReqClient {
20.
21.     public void connect(int port, String host) throws E
22.     // 配置客户端NIO线程组
23.     EventLoopGroup group = new NioEventLoopGroup();
24.     try {
25.         Bootstrap b = new Bootstrap();
26.         b.group(group).channel(NioSocketChannel.class)
27.         .option(ChannelOption.TCP_NODELAY, true)
```

```
28.         .handler(new ChannelInitializer<SocketChann
29.         @Override
30.         public void initChannel(SocketChannel ch)
31.             throws Exception {
32.             ch.pipeline().addLast(

33.                 new ObjectDecoder(1024, ClassResolv

34.                 .cacheDisabled(this.getClass()

35.                 .getClassLoader())));

36.         ch.pipeline().addLast(new ObjectEncoder

37.         ch.pipeline().addLast(new SubReqClientH
```

```
38.         }
39.     });
40.
41.     // 发起异步连接操作
42.     ChannelFuture f = b.connect(host, port).sync();
43.
44.     // 等待客户端链路关闭
45.     f.channel().closeFuture().sync();
46. } finally {
47.     // 优雅退出，释放NIO线程组
48.     group.shutdownGracefully();
49. }
50. }
51.
52. /**
53.  * @param args
54.  * @throws Exception
55.  */
56. public static void main(String[] args) throws Excep
57. int port = 8080;
58. if (args != null && args.length > 0) {
59.     try {
60.         port = Integer.valueOf(args[0]);
61.     } catch (NumberFormatException e) {
62.         // 采用默认值
63.     }
```

```
64.         }
65.         new SubReqClient().connect(port, "127.0.0.1");
66.     }
67. }
```

第32行，我们禁止对类加载器进行缓存，它在基于OSGi的动态模块化编程中经常使用。由于OSGi的bundle可以进行热部署和热升级，当某个bundle升级后，它对应的类加载器也将一起升级，因此在动态模块化编程过程中，很少对类加载器进行缓存，因为它随时可能会发生变化。

下面继续看下SubReqClientHandler的实现。

代码清单7-6 Netty Java序列化 产品订购客户端
SubReqClientHandler

```
12.     public class SubReqClientHandler extends ChannelHandler
13.
14.         /**
15.          * Creates a client-side handler.
16.          */
17.         public SubReqClientHandler() {
18.         }
19.
20.         @Override
21.         public void channelActive(ChannelHandlerContext ctx
22.             for (int i = 0; i < 10; i++) {
```

```
23.         ctx.write(subReq(i));
24.     }
25.     ctx.flush();
26. }
27.
28.     private SubscribeReq subReq(int i) {
29.         SubscribeReq req = new SubscribeReq();
30.         req.setAddress("南京市江宁区方山国家地质公园");
31.         req.setPhoneNumber("138xxxxxxxx");
32.         req.setProductName("Netty 权威指南");
33.         req.setSubReqID(i);
34.         req.setUserName("Lilinfeng");
35.         return req;
36.     }
37.
38.     @Override
39.     public void channelRead(ChannelHandlerContext ctx,
40.         throws Exception {
41.         System.out.println("Receive server response : [" +
42.     }
43.
44.     @Override
45.     public void channelReadComplete(ChannelHandlerContext conte
46.         ctx.flush();
47.     }
48.
49.     @Override
```



```
50.         public void exceptionCaught(ChannelHandlerContext c
51.             cause.printStackTrace();
52.             ctx.close();
53.         }
54.     }
```

第22～25行，在链路激活的时候循环构造10条订购请求消息，最后一次性地发送给服务端。

由于对象解码器已经对订购请求应答消息进行了自动解码，因此，SubReqClientHandler接收到的消息已经是解码成功后的订购应答消息。

下面的小节将执行我们前面开发的订购请求客户端和服务端，看下执行结果是否符合设计预期。

7.3 运行结果

运行**Java**序列化例程

首先运行服务端，然后运行客户端，运行结果如下。

服务端运行结果如下。

```
log4j:WARN No appenders could be found for logger (io.netty.u
log4j:WARN Please initialize the log4j system properly.
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
Service accept client subscribe req : [SubscribeReq [subReqID
```

尽管客户端一次批量发送了10条订购请求消息，TCP会对请求消息进行粘包和拆包，但是并没有影响最终的运行结果：服务端成功收到了10条订购请求消息，与客户端发送的一致。

客户端运行结果如下。

```
log4j:WARN No appenders could be found for logger (io.netty.u
log4j:WARN Please initialize the log4j system properly.
Receive server response : [SubscribeResp [subReqID=0, respCod
Receive server response : [SubscribeResp [subReqID=1, respCod
Receive server response : [SubscribeResp [subReqID=2, respCod
Receive server response : [SubscribeResp [subReqID=3, respCod
Receive server response : [SubscribeResp [subReqID=4, respCod
Receive server response : [SubscribeResp [subReqID=5, respCod
Receive server response : [SubscribeResp [subReqID=6, respCod
Receive server response : [SubscribeResp [subReqID=7, respCod
Receive server response : [SubscribeResp [subReqID=8, respCod
Receive server response : [SubscribeResp [subReqID=9, respCod
```

客户端接收到了10条订购应答消息，Netty的ObjectEncoder编码器可以自动对订购应答消息进行序列化，然后发送给客户端，客户端的ObjectDecoder对码流进行反序列化，获得订购请求应答消息。

7.4 总结

本章介绍了如何利用Netty提供的ObjectEncoder编码器和ObjectDecoder解码器实现对普通POJO对象的序列化。通过订购图书例程，我们学习了服务端和客户端的开发，并且模拟了TCP粘包/拆包场景，对运行结果进行了分析。

通过使用Netty的Java序列化编解码handler，用户通过短短的几行代码，就能完成POJO的序列化和反序列化。在业务处理handler中，用户只需要将精力聚焦在业务逻辑的实现上，不需要关心底层的编解码细节，这极大地提升了开发效率。

下一章我们继续学习谷歌的Protobuf，看在Netty中如何使用Protobuf实现对POJO对象的自动编解码。

第8章 Google Protobuf编解码

Google的Protobuf在业界非常流行，很多商业项目选择Protobuf作为编解码框架，这里一起回顾一下Protobuf的优点。

- (1) 在谷歌内部长期使用，产品成熟度高；
- (2) 跨语言，支持多种语言，包括C++、Java和Python；
- (3) 编码后的消息更小，更加有利于存储和传输；
- (4) 编解码的性能非常高；
- (5) 支持不同协议版本的前向兼容；
- (6) 支持定义可选和必选字段。

本章主要包括：

- Protobuf的入门
- 开发支持Protobuf的Netty服务端
- 开发支持Protobuf的Netty客户端
- 运行基于Netty开发的Protobuf例程

8.1 Protobuf的入门

Protobuf是一个灵活、高效、结构化的数据序列化框架，相比于XML等传统的序列化工具，它更小，更快，更简单。Protobuf支持数据结构化一次可以到处使用，甚至跨语言使用，通过代码生成工具可以自动生成不同语言版本的源代码，甚至可以在使用不同版本的数据结构进程间进行数据传递，实现数据结构的前向兼容。

下面我们通过一个简单的例程来学习如何使用Protobuf对POJO对象进行编解码，然后，我们以这个例程为基础，学习如何在Netty中对POJO对象进行Protobuf编解码，并在两个进程之间进行通信和数据交换。

8.1.1 Protobuf开发环境搭建

首先下载Protobuf的最新windows版本，网址如下：

<http://code.google.com/p/protobuf/downloads/detail?name=protoc-2.5.0-win32.zip&can=2&q=>

对下载的protoc-2.5.0-win32.zip进行解压，解压后的目录如图8-1所示。

图8-1 Protobuf解压后的目录

protoc.exe工具主要根据.proto文件生成代码，下面我们以第7章的订购例程为例，定义SubscribeReq.proto和SubscribeResp.proto，数据文件定义如下。

- SubscribeReq.proto，如图8-2所示。

图8-2 SubscribeReq.proto文件定义

- SubscribeResp.proto，如图8-3所示。

图8-3 SubscribeResp.proto文件定义

通过protoc.exe命令行生成Java代码，命令行如图8-4所示。

图8-4 通过protoc.exe工具生成源代码

将生成的POJO代码SubscribeReqProto.java和SubscribeRespProto.java复制到对应的Eclipse工程中，目录示例如图8-5所示。

图8-5 将生成的POJO代码拷贝到源工程中

我们发现代码编译出错，原因是缺少protobuf-java-2.5.0.jar包，从Google官网下载后将其复制到lib目录后编译到引用类库中，如图8-6所示。

图8-6 编译Protobuf工程

到此为止，Protobuf开发环境已经搭建完毕，接下来将进行示例开发。

8.1.2 Protobuf编解码开发

Protobuf的类库使用比较简单，下面我们就通过对SubscribeReqProto进行编解码来介

绍Protobuf的使用。

代码清单8-1 Protobuf入门TestSubscribeReqProto

```
12. public class TestSubscribeReqProto {
13.
14.     private static byte[] encode(SubscribeReqProto.Subscri
15.         return req.toByteArray();
16.     }
17.
18.     private static SubscribeReqProto.SubscribeReq decode(b
19.         throws InvalidProtocolBufferException {
20.         return SubscribeReqProto.SubscribeReq.parseFrom(body);
21.     }
22.
23.     private static SubscribeReqProto.SubscribeReq createSu
24.         SubscribeReqProto.SubscribeReq.Builder builder = Subsc
25.             .newBuilder();
26.         builder.setSubReqID(1);
27.         builder.setUserName("Lilinfeng");
28.         builder.setProductName("Netty Book");
29.         List<String> address = new ArrayList<>();
30.         address.add("NanJing YuHuaTai");
31.         address.add("BeiJing LiuLiChang");
32.         address.add("ShenZhen HongShuLin");
33.         builder.addAllAddress(address);
34.         return builder.build();
```



```
35.     }
36.
37.     /**
38.      * @param args
39.      * @throws InvalidProtocolBufferException
40.      */
41.     public static void main(String[] args)
42.         throws InvalidProtocolBufferException {
43.         SubscribeReqProto.SubscribeReq req = createSubscribeRe
44.         System.out.println("Before encode : " + req.toString())
45.         SubscribeReqProto.SubscribeReq req2 = decode(encode(re
46.         System.out.println("After decode : " + req2.toString())
47.         System.out.println("Assert equal : --> " + req2.equals
48.     }
49. }
```

首先我们看如何创建SubscribeReqProto.SubscribeReq实例，第24行通过SubscribeReqProto.SubscribeReq的静态方法newBuilder创建SubscribeReqProto.SubscribeReq的Builder实例，通过Builder构建器对SubscribeReq的属性进行设置，对于集合类型，通过addAllXXX()方法可以将集合对象设置到对应的属性中。

编码时通过调用SubscribeReqProto.SubscribeReq实例的toByteArray即可将SubscribeReq编码为byte数组，使用非常方便。

解码时通过SubscribeReqProto.SubscribeReq的静态方法parseFrom将二进制byte数组解码为原始的对象。

由于Protobuf支持复杂POJO对象编解码，所以代码都是通过工具自动生成，相比于传统的POJO对象的赋值操作，其使用略微复杂一些，但是习惯之后也不会带来额外的工作量，主要差异还是编程习惯的不同。

Protobuf的编解码接口非常简单和实用，但是功能和性能却非常强大，这也是它流行的一个重要原因。

下个小节我们将执行TestSubscribeReqProto，看它的功能是否正常。

8.1.3 运行Protobuf例程

我们运行上一小节编写的TestSubscribeReqProto程序，看经过编解码后的对象是否和编码之前的初始对象等价，代码执行结果如图8-7所示。

图8-7 Protobuf编解码运行结果

运行结果表明，经过Protobuf编解码后，生成的SubscribeReqProto.SubscribeReq与编码前原始的SubscribeReqProto.SubscribeReq等价。

至此，我们已经学会了如何搭建Protobuf的开发和运行环境，并初步掌握了Protobuf的编解码接口的使用方法，而且通过实际demo的开发和运行巩固了所学的知识。从下个小节开始，我们将学习使用Netty的Protobuf编解码框架。

8.2 Netty的Protobuf服务端开发

我们仍旧以第7章的例程作为demo进行学习，看看如何开发出一个Protobuf版本的图书订购程序。

8.2.1 Protobuf版本的图书订购服务端开发

对SubReqServer进行升级，代码如下。

代码清单8-2 Protobuf版本图书订购代码SubReqServer

```
20. public class SubReqServer {
21.     public void bind(int port) throws Exception {
22.         // 配置服务端的NIO 线程组
23.         EventLoopGroup bossGroup = new NioEventLoopGroup();
24.         EventLoopGroup workerGroup = new NioEventLoopGroup();
25.         try {
26.             ServerBootstrap b = new ServerBootstrap();
27.             b.group(bossGroup, workerGroup)
28.                 .channel(NioServerSocketChannel.class)
29.                 .option(ChannelOption.SO_BACKLOG, 100)
30.                 .handler(new LoggingHandler(LogLevel.INFO))
31.                 .childHandler(new ChannelInitializer<SocketChan
32.                 @Override
33.                 public void initChannel(SocketChannel ch) {
34.                     ch.pipeline().addLast(
```

35. `new ProtobufVarint32FrameDecoder());`

36. `ch.pipeline().addLast(`

37. `new ProtobufDecoder(`

38. `SubscribeReqProto.SubscribeReq`

39. `.getDefaultInstance()));`

40. `ch.pipeline().addLast(`

41. `new ProtobufVarint32LengthFieldPrepender`

```
42.          ch.pipeline().addLast(new ProtobufEncoder())

43.          ch.pipeline().addLast(new SubReqServerHandle

44.      }
45.  });
46.
47.      // 绑定端口，同步等待成功
48.      ChannelFuture f = b.bind(port).sync();
49.
50.      // 等待服务端监听端口关闭
51.      f.channel().closeFuture().sync();
52.  } finally {
53.      // 优雅退出，释放线程池资源
54.      bossGroup.shutdownGracefully();
55.      workerGroup.shutdownGracefully();
56.  }
57.  }
58.
59.  public static void main(String[] args) throws Exception
```

```
60.    int port = 8080;
61.    if (args != null && args.length > 0) {
62.        try {
63.            port = Integer.valueOf(args[0]);
64.        } catch (NumberFormatException e) {
65.            // 采用默认值
66.        }
67.    }
68.    new SubReqServer().bind(port);
69. }
70. }
```

第34行首先向ChannelPipeline添加ProtobufVarint32FrameDecoder，它主要用于半包处理，随后继续添加ProtobufDecoder解码器，它的参数是com.google.protobuf.MessageLite，实际上就是要告诉ProtobufDecoder需要解码的目标类是什么，否则仅仅从字节数组中是无法判断出要解码的目标类型信息的。

下面我们继续看SubReqServerHandler的实现。

代码清单8-3 Protobuf版本图书订购代码SubReqServerHandler

```
11. @Sharable
12. public class SubReqServerHandler extends ChannelHandlerAd
13.
14.     @Override
15.     public void channelRead(ChannelHandlerContext ctx, Obj
```

```
16.         throws Exception {
17.     SubscribeReqProto.SubscribeReq req = (SubscribeReqProt
18.     if ("Lilinfeng".equalsIgnoreCase(req.getUserName())) {
19.         System.out.println("Service accept client subscribe
20.             + req.toString() + "]);
21.         ctx.writeAndFlush(resp(req.getSubReqID()));
22.     }
23. }
24.
25. private SubscribeRespProto.SubscribeResp resp(int subRe
26.     SubscribeRespProto.SubscribeResp.Builder builder = Sub
27.     .newBuilder();
28.     builder.setSubReqID(subReqID);
29.     builder.setRespCode(0);
30.     builder.setDesc("Netty book order succeed, 3 days late
31.     return builder.build();
32. }
33.
34. @Override
35. public void exceptionCaught(ChannelHandlerContext ctx,
36.     cause.printStackTrace();
37.     ctx.close();// 发生异常，关闭链路
38. }
39. }
```

由于ProtobufDecoder已经对消息进行了自动解码，因此接收到的订

购请求消息可以直接使用。对用户名进行校验，校验通过后构造应答消息返回给客户端，由于使用了ProtobufEncoder，所以不需要对SubscribeRespProto.SubscribeResp进行手工编码。

下个小节我们继续看客户端的代码实现。

8.2.2 Protobuf版本的图书订购客户端开发

与第7章的demo类似，唯一不同的就是订购请求消息使用Protobuf进行消息编解码。

代码清单8-4 Protobuf版本图书订购代码SubReqClient

```
20. public class SubReqClient {
21.
22.     public void connect(int port, String host) throws Exce
23.         // 配置客户端NIO 线程组
24.         EventLoopGroup group = new NioEventLoopGroup();
25.         try {
26.             Bootstrap b = new Bootstrap();
27.             b.group(group).channel(NioSocketChannel.class)
28.                 .option(ChannelOption.TCP_NODELAY, true)
29.                 .handler(new ChannelInitializer<SocketChannel>()
30.                     @Override
31.                     public void initChannel(SocketChannel ch)
32.                         throws Exception {
33.                             ch.pipeline().addLast(
```


34. `new ProtobufVarint32FrameDecoder());`

35. `ch.pipeline().addLast(`

36. `new ProtobufDecoder(`

37. `SubscribeRespProto.SubscribeResp`

38. `.getDefaultInstance()));`

39. `ch.pipeline().addLast(`

40. `new ProtobufVarint32LengthFieldPrepender`

```
41.                ch.pipeline().addLast(new ProtobufEncoder())

42.                ch.pipeline().addLast(new SubReqClientHandle

43.            }
44.        });
45.
46.        // 发起异步连接操作
47.        ChannelFuture f = b.connect(host, port).sync();
48.
49.        // 等待客户端链路关闭
50.        f.channel().closeFuture().sync();
51.    } finally {
52.        // 优雅退出，释放NIO 线程组
53.        group.shutdownGracefully();
54.    }
55. }
56.
57. /**
58.  * @param args
```

```
59.     * @throws Exception
60.     */
61.     public static void main(String[] args) throws Exception
62.     {
63.         int port = 8080;
64.         if (args != null && args.length > 0) {
65.             try {
66.                 port = Integer.valueOf(args[0]);
67.             } catch (NumberFormatException e) {
68.                 // 采用默认值
69.             }
70.         }
71.         new SubReqClient().connect(port, "127.0.0.1");
72.     }
```

需要指出的是客户端需要解码的对象是订购响应，所以第37～38行使用SubscribeResp

Proto.SubscribeResp的实例做入参。

代码清单8-5 Protobuf版本图书订购代码 SubReqClientHandler

```
13. public class SubReqClientHandler extends ChannelHandlerAd
14. {
15.     /**
16.      * Creates a client-side handler.
17.      */
```

```
18.     public SubReqClientHandler() {
19.     }
20.
21.     @Override
22.     public void channelActive(ChannelHandlerContext ctx) {
23.         for (int i = 0; i < 10; i++) {
24.             ctx.write(subReq(i));
25.         }
26.         ctx.flush();
27.     }
28.
29.     private SubscribeReqProto.SubscribeReq subReq(int i) {
30.         SubscribeReqProto.SubscribeReq.Builder builder = Subsc
31.             .newBuilder();
32.         builder.setSubReqID(i);
33.         builder.setUserName("Lilinfeng");
34.         builder.setProductName("Netty Book For Protobuf");
35.         List<String> address = new ArrayList<>();
36.         address.add("NanJing YuHuaTai");
37.         address.add("BeiJing LiuLiChang");
38.         address.add("ShenZhen HongShuLin");
39.         builder.addAllAddress(address);
40.         return builder.build();
41.     }
42.
43.     @Override
44.     public void channelRead(ChannelHandlerContext ctx, Obj
```

```
45.         throws Exception {
46.     System.out.println("Receive server response : [" + msg
47.     }
48.
49.     @Override
50.     public void channelReadComplete(ChannelHandlerContext
51.     ctx.flush();
52.     }
53.
54.     @Override
55.     public void exceptionCaught(ChannelHandlerContext ctx,
56.     cause.printStackTrace();
57.     ctx.close();
58.     }
59. }
```

客户端接收到服务端的应答消息之后会直接打印，按照设计，应该打印10次。下面我们就测试下Protobuf的服务端和客户端，看它是否能正常运行。

8.2.3 Protobuf版本的图书订购程序功能测试

分别运行服务端和客户端，运行结果如下。

服务端运行结果如下。

```
Service accept client subscribe req : [subReqID: 0
```

```
userName: "Lilinfeng"  
productName: "Netty Book For Protobuf"  
address: "NanJing YuHuaTai"  
address: "BeiJing LiuLiChang"  
address: "ShenZhen HongShuLin"  
]
```

```
Service accept client subscribe req : [subReqID: 1  
userName: "Lilinfeng"  
productName: "Netty Book For Protobuf"  
address: "NanJing YuHuaTai"  
address: "BeiJing LiuLiChang"  
address: "ShenZhen HongShuLin"  
]
```

```
Service accept client subscribe req : [subReqID: 2  
userName: "Lilinfeng"  
productName: "Netty Book For Protobuf"  
address: "NanJing YuHuaTai"  
address: "BeiJing LiuLiChang"  
address: "ShenZhen HongShuLin"  
]
```

```
Service accept client subscribe req : [subReqID: 3  
userName: "Lilinfeng"  
productName: "Netty Book For Protobuf"  
address: "NanJing YuHuaTai"  
address: "BeiJing LiuLiChang"  
address: "ShenZhen HongShuLin"  
]
```

Service accept client subscribe req : [subReqID: 4
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 5
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 6
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 7
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"

```
]
Service accept client subscribe req : [subReqID: 8
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]
Service accept client subscribe req : [subReqID: 9
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]
```

客户端运行结果如下。

```
Receive server response : [subReqID: 0
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]
Receive server response : [subReqID: 1
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]
```


Receive server response : [subReqID: 2
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]

Receive server response : [subReqID: 3
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]

Receive server response : [subReqID: 4
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]

Receive server response : [subReqID: 5
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]

Receive server response : [subReqID: 6
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]

Receive server response : [subReqID: 7
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]

Receive server response : [subReqID: 8
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de

```
]
Receive server response : [subReqID: 9
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the de
]
```

运行结果表明，我们基于Netty Protobuf编解码框架开发的图书订购程序可以正常工作。利用Netty提供的Protobuf编解码能力，我们在不需要了解Protobuf实现和使用细节的情况下就能轻松支持Protobuf编解码，可以方便地实现跨语言的远程服务调用和与周边的异构系统进行通信对接。

8.3 Protobuf的使用注意事项

ProtobufDecoder仅仅负责解码，它不支持读半包。因此，在ProtobufDecoder前面，一定要有能够处理读半包的解码器，有三种方式可以选择。

- 使用Netty提供的ProtobufVarint32FrameDecoder，它可以处理半包消息；
- 继承Netty提供的通用半包解码器LengthFieldBasedFrameDecoder；
- 继承ByteToMessageDecoder类，自己处理半包消息。

如果你只使用ProtobufDecoder解码器而忽略对半包消息的处理，程序是不能正常工作的。以前面的图书订购为例对服务端代码进行修改，注释掉ProtobufVarint32FrameDecoder，代码修改如图8-8所示。

图8-8 注释掉ProtobufVarint32FrameDecoder

运行程序，结果如图8-9所示，运行出错。

图8-9 注释掉ProtobufVarint32FrameDecoder运行出错

8.4 总结

本章首先介绍了Protobuf的入门知识，通过一个简单的样例代码开发让读者熟悉了如何使用Protobuf对POJO对象进行编解码；在掌握了Protobuf的基础知识之后，讲解如何使用Netty的Protobuf编解码框架进行客户端和服务端的开发；最后，对Protobuf解码器的使用陷阱进行了说明，并给出了正确的使用建议。

在下一章中，我们继续学习另一种序列化技术——JBoss的Marshallng序列化框架，它是JBoss内部使用的序列化框架，Netty提供了Marshallng编码和解码器，方便用户在Netty中使用Marshallng。第9章适用于对Marshallng框架感兴趣的读者，如果你想直接学习后面的知识，也可以跳过第9章。

第9章 JBoss Marshalling编解码

JBoss Marshalling是一个Java对象序列化包，对JDK默认的序列化框架进行了优化，但又保持跟java.io.Serializable接口的兼容，同时增加了一些可调的参数和附加的特性，这些参数和特性可通过工厂类进行配置。

本章主要包括：

- Marshalling开发环境准备
- Netty的Marshalling服务端开发
- Netty的Marshalling客户端开发
- 运行Marshalling例程

9.1 Marshalling开发环境准备

首先下载相关的Marshalling类库，由于我们只是用到了它的序列化类库，因此，只需要下载jboss-marshalling-1.3.0和jboss-marshalling-serial-1.3.0类库即可，下载网址如下：

<https://www.jboss.org/jbossmarshalling/downloads>

在页面上选择JBoss Marshalling API和JBoss Marshalling Serial Protocol进行下载，如图9-1所示。

图9-1 Marshalling类库下载

将下载的类库build到classpath中，如图9-2所示。

图9-2 将Marshalling添加到引用类库中

Marshalling开发环境搭建完成之后，我们开始学习Marshalling服务端的开发。

9.2 Netty的Marshalling服务端开发

首先定义POJO对象，由于JBoss的Marshalling完全兼容JDK序列化，因此我们继续使用第7章定义的SubscribeReq和SubscribeResp对象。通过JBoss提供的序列化API，对SubscribeReq和SubscribeResp进行编解码。由于Netty对JBoss的编解码类库进行了封装，下面通过图书订购的实例看看如何使用Netty的Marshalling编解码类对消息进行序列化和反序列化。

服务端开发示例

首先看下服务端启动类的开发，代码如下。

代码清单9-1 Marshalling版本图书订购代码 SubReqServer

```
18. public class SubReqServer {
19.     public void bind(int port) throws Exception {
20.         // 配置服务端的NIO线程组
21.         EventLoopGroup bossGroup = new NioEventLoopGroup();
22.         EventLoopGroup workerGroup = new NioEventLoopGroup(
23.             try {
24.                 ServerBootstrap b = new ServerBootstrap();
25.                 b.group(bossGroup, workerGroup)
26.                     .channel(NioServerSocketChannel.class)
27.                     .option(ChannelOption.SO_BACKLOG, 100)
28.                     .handler(new LoggingHandler(LogLevel.INFO))
```

```
29.         .childHandler(new ChannelInitializer<Socket  
30.         @Override  
31.         public void initChannel(SocketChannel ch) {  
32.             ch.pipeline().addLast(  

```

```
33.             MarshallingCodeCFactory
```

```
34.             .buildMarshallingDecoder());
```

```
35.             ch.pipeline().addLast(  

```

```
36.             MarshallingCodeCFactory
```

```
37.             .buildMarshallingEncoder());
```



```
38.             ch.pipeline().addLast(new SubReqServerH
39.             }
40.             });
41.
42.             // 绑定端口，同步等待成功
43.             ChannelFuture f = b.bind(port).sync();
44.
45.             // 等待服务端监听端口关闭
46.             f.channel().closeFuture().sync();
47.         } finally {
48.             // 优雅退出，释放线程池资源
49.             bossGroup.shutdownGracefully();
50.             workerGroup.shutdownGracefully();
51.         }
52.     }
53.
54.     public static void main(String[] args) throws Excep
55.     int port = 8080;
56.     if (args != null && args.length > 0) {
57.         try {
58.             port = Integer.valueOf(args[0]);
59.         } catch (NumberFormatException e) {
60.             // 采用默认值
61.         }
62.     }
63.     new SubReqServer().bind(port);
64. }
```

65. }

第32~34行通过MarshallingCodeCFactory工厂类创建了MarshallingDecoder解码器，并将其加入到ChannelPipeline中；第35~37行通过工厂类创建MarshallingEncoder编码器，并添加到ChannelPipeline中。

下面继续看MarshallingCodeCFactory是如何实现的，代码如下。

代码清单9-2 Marshalling版本图书订购代码
MarshallingCodeCFactory

```
18. public final class MarshallingCodeCFactory {
19.
20.     /**
21.      * 创建Jboss Marshalling解码器MarshallingDecoder
22.      *
23.      * @return
24.      */
25.     public static MarshallingDecoder buildMarshallingDe
26.     final MarshallerFactory marshallerFactory = Marshal

27.         .getProvidedMarshallerFactory("serial");
```

28. *final MarshallingConfiguration configuration = new*

29. *configuration.setVersion(5);*

30. *UnmarshallerProvider provider = new DefaultUnmarsha*

31. *marshallerFactory, configuration);*

32. *MarshallingDecoder decoder = new MarshallingDecoder*

33. *return decoder;*

34. *}*

35.

```
36.      /**
37.      * 创建Jboss Marshalling编码器MarshallingEncoder
38.      *
39.      * @return
40.      */
41.      public static MarshallingEncoder buildMarshallingEn
42.      final MarshallerFactory marshallerFactory = Marshal

43.          .getProvidedMarshallerFactory("serial");

44.      final MarshallingConfiguration configuration = new

45.      configuration.setVersion(5);

46.      MarshallerProvider provider = new DefaultMarshaller

47.          marshallerFactory, configuration);
```

```
48.         MarshallingEncoder encoder = new MarshallingEncoder

49.         return encoder;

50.     }
51. }
```

第26~27行首先通过Marshalling工具类的getProvidedMarshallerFactory静态方法获取MarshallerFactory实例，参数“serial”表示创建的是Java序列化工厂对象，它由jboss-marshalling-serial-1.3.0.CR9.jar提供。

第28行创建了MarshallingConfiguration对象，将它的版本号设置为5，然后根据MarshallerFactory和MarshallingConfiguration创建UnmarshallerProvider实例，最后通过构造函数创建Netty的MarshallingDecoder对象，它有两个参数，分别是UnmarshallerProvider和单个消息序列化后的最大长度。

第42~45行同样是构造MarshallerFactory和MarshallingConfiguration，第46行创建MarshallerProvider对象，它用于

创建Netty提供的MarshallingEncoder实例， MarshallingEncoder用于将实现序列化接口的POJO对象序列化为二进制数组。

由于SubReqServerHandler的实现与第8章例程中的SubReqServerHandler实现完全相同，因此这里不再给出源码。

9.3 Netty的Marshalling客户端开发

首先由客户端发送订购请求消息，为了测试TCP粘包/拆包是否能被正确处理，采取连续发送10条请求消息的策略。在客户端的ChannelPipeline中添加MarshallingEncoder编码器对POJO对象进行编码。接收服务端应答消息的时候需要对经过Marshalling序列化后的码流进行解码，因此也需要添加MarshallingDecoder，下面我们看下客户端代码的具体实现。

客户端开发示例

首先看客户端启动类。

代码清单9-3 Marshalling版本图书订购代码 SubReqClient

```
16. public class SubReqClient {
17.
18.     public void connect(int port, String host) throws E
19.         // 配置客户端NIO线程组
20.         EventLoopGroup group = new NioEventLoopGroup();
21.         try {
22.             Bootstrap b = new Bootstrap();
23.             b.group(group).channel(NioSocketChannel.class)
24.                 .option(ChannelOption.TCP_NODELAY, true)
25.                 .handler(new ChannelInitializer<SocketChann
26.                 @Override
```

```
27.         public void initChannel(SocketChannel ch)
28.             throws Exception {
29.             ch.pipeline().addLast(
```

```
30.                 MarshallingCodeCFactory
```

```
31.                 .buildMarshallingDecoder());
```

```
32.             ch.pipeline().addLast(
```

```
33.                 MarshallingCodeCFactory
```

```
34.                 .buildMarshallingEncoder());
```

```
35.             ch.pipeline().addLast(new SubReqClientH
```



```
36.         }
37.         });
38.
39.         // 发起异步连接操作
40.         ChannelFuture f = b.connect(host, port).sync();
41.
42.         // 等待客户端链路关闭
43.         f.channel().closeFuture().sync();
44.     } finally {
45.         // 优雅退出，释放NIO线程组
46.         group.shutdownGracefully();
47.     }
48. }
49.
50. /**
51.  * @param args
52.  * @throws Exception
53.  */
54. public static void main(String[] args) throws Excep
55. int port = 8080;
56. if (args != null && args.length > 0) {
57.     try {
58.         port = Integer.valueOf(args[0]);
59.     } catch (NumberFormatException e) {
60.         // 采用默认值
61.     }
62. }
```

```
63.         new SubReqClient().connect(port, "127.0.0.1");
64.     }
65. }
```

第29～34行分别创建Marshalling编码器和解码器，并将其添加到ChannelPipeline中。

SubReqClientHandler相比于第8章的例程，仅仅修改了订购应答消息的产品名称，具体修改如图9-3所示。

图9-3 构造订购成功应答消息

9.4 运行Marshalling客户端和服务端例程

分别运行图书订购服务端和客户端例程，运行结果如下。

服务端运行结果如下。

```
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
Service accept client subscrib req : [SubscribeReq [subReqID=
```

服务端共收到了10条客户端请求消息，subReqID为从0到9，与客户端发送的订购请求消息完全一致，服务端运行结果正确。

客户端运行结果如下。

```
Receive server response : [SubscribeResp [subReqID=0, respCod
Receive server response : [SubscribeResp [subReqID=1, respCod
Receive server response : [SubscribeResp [subReqID=2, respCod
Receive server response : [SubscribeResp [subReqID=3, respCod
```

```
Receive server response : [SubscribeResp [subReqID=4, respCod
Receive server response : [SubscribeResp [subReqID=5, respCod
Receive server response : [SubscribeResp [subReqID=6, respCod
Receive server response : [SubscribeResp [subReqID=7, respCod
Receive server response : [SubscribeResp [subReqID=8, respCod
Receive server response : [SubscribeResp [subReqID=9, respCod
```

客户端成功收到了服务端返回的10条应答消息，subReqID为从0到9，与服务端发送的应答消息完全一致，测试表明客户端运行结果正确。

由于我们模拟了TCP的粘包/拆包场景，但是程序的运行结果仍然正确，说明Netty的Marshalling编解码器支持半包和粘包的处理，对于开发者而言，只需要正确地将Marshalling编码器和解码器加入到ChannelPipeline中，就能实现对Marshalling序列化的支持。

利用Netty的Marshalling编解码器，可以轻松地开发出与JBoss内部模块进行远程通信的程序，而且支持异步非阻塞，这无疑降低了基于Netty开发的应用程序与JBoss内部模块对接的难度。

9.5 总结

本章介绍了如何使用Netty的Marshalling编码器和解码器对POJO对象进行序列化。通过使用Netty的Marshalling编解码器，我们可以轻松地开发出支持JBoss Marshalling序列化的客户端和服务端程序，方便地对接JBoss的内部模块，同时也有利于对已有使用Jboss Marshalling框架做通信协议的模块的桥接和重用。

从下一章开始，我们将学习如何使用Netty的HTTP协议栈进行HTTP服务端和客户端的开发。由于HTTP协议目前仍然是各个行业主流的系统间通信协议，因此，Netty的HTTP协议栈的应用空间非常广泛。

高级篇 **Netty**多协议开发和应用

第10章 HTTP协议开发应用

第11章 WebSocket协议开发

第12章 UDP协议开发

第13章 文件传输

第14章 私有协议栈开发

第10章 HTTP协议开发应用

HTTP（超文本传输协议）协议是建立在TCP传输协议之上的应用层协议，它的发展是万维网协会和Internet工作小组IETF合作的结果。HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过多年的使用和发展，得到了不断地完善和扩展。

由于HTTP协议是目前Web开发的主流协议，基于HTTP的应用非常广泛，因此，掌握HTTP的开发非常重要，本章将重点介绍如何基于Netty的HTTP协议栈进行HTTP服务端和客户端开发。由于Netty的HTTP协议栈是基于Netty的NIO通信框架开发的，因此，Netty的HTTP协议也是异步非阻塞的。

本章主要包括：

- HTTP协议介绍
- Netty HTTP服务端入门开发
- HTTP + XML应用开发
- HTTP附件处理

10.1 HTTP协议介绍

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。

HTTP协议的主要特点如下。

- 支持Client/Server模式；
- 简单——客户向服务器请求服务时，只需指定服务URL，携带必要的请求参数或者消息体；
- 灵活——HTTP允许传输任意类型的数据对象，传输的内容类型由HTTP消息头中的Content-Type加以标记；
- 无状态——HTTP协议是无状态协议，无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要之前的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快，负载较轻。

10.1.1 HTTP协议的URL

HTTP URL（URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息）的格式如下。

```
http://host[":"port][abs_path]
```

其中，http表示要通过HTTP协议来定位网络资源；host表示合法的Internet主机域名或者IP地址；port指定一个端口号，为空则使用默认端口80；abs_path指定请求资源的URI，如果URL中没有给出abs_path，那

么当它作为请求URI时，必须以“/”的形式给出，通常这点工作浏览器会自动帮我们完成。

10.1.2 HTTP请求消息（HttpRequest）

HTTP请求由三部分组成，具体如下。

- HTTP请求行；
- HTTP消息头；
- HTTP请求正文。

请求行以一个方法符开头，以空格分开，后面跟着请求的URI和协议的版本，格式为：Method Request-URI HTTP-Version CRLF。

其中Method表示请求方法，Request-URI是一个统一资源标识符，HTTP-Version表示请求的HTTP协议版本，CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

请求方法有多种，各方法的作用如下。

- GET：请求获取Request-URI所标识的资源；
- POST：在Request-URI所标识的资源后附加新的提交数据；
- HEAD：请求获取由Request-URI所标识的资源的响应消息报头；
- PUT：请求服务器存储一个资源，并用Request-URI作为其标识；
- DELETE：请求服务器删除Request-URI所标识的资源；
- TRACE：请求服务器回送收到的请求信息，主要用于测试或诊断；
- CONNECT：保留将来使用；
- OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项

和需求。

GET方法： 以在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用GET方法向服务器获取资源。例如，我们直接在浏览器中输入 `http://localhost:8080/netty-5.0.0`，如图10-1所示。

图10-1 通过浏览器访问Netty HTTP服务端

通过服务端抓包，打印HTTP请求消息头，内容如下。

```
GET /netty5.0 HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/37.0.2062.120 Safari/537.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.7
Accept-Charset: GBK,utf-8;q=0.7,*;q=0.3
Content-Length: 0
```

我们可以看到第一行请求行使用的是GET方法。

POST方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。GET和POST的主要区别如下。

(1) 根据HTTP规范，GET用于信息获取，而且应该是安全的和幂等的；POST则表示可能修改变服务器上的资源的请求。

(2) GET提交，请求的数据会附在URL之后，就是把数据放置在请求行（request line）中，以“？”分割URL和传输数据，多个参数用“&”连接；而POST提交会把提交的数据放置在HTTP消息的包体中，数据不会在地址栏中显示出来。

(3) 传输数据的大小不同。特定浏览器和服务端对URL长度有限制，例如IE对URL长度的限制是2083字节（2K+35），因此GET携带的参数的长度会受到浏览器的限制；而POST由于不是通过URL传值，理论上数据长度不会受限。

(4) 安全性。POST的安全性要比GET的安全性高。比如通过GET提交数据，用户名和密码将明文出现在URL上。因为1) 登录页面有可能被浏览器缓存；2) 其他人查看浏览器的历史记录，那么别人就可以拿到你的账号和密码了。除此之外，使用GET提交数据还可能会造成Cross-site request forgery攻击。POST提交的内容由于在消息体中传输，因此不存在上述安全问题。

请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。常用的请求报头如表10-1所示。

表10-1 HTTP的部分请求消息头列表

HTTP请求消息体是可选的，比较常用的HTTP+XML协议就是通过HTTP请求和响应消息体来承载XML信息的。

10.1.3 HTTP响应消息（HttpResponse）

处理完HTTP客户端的请求之后，HTTP服务端返回响应消息给客户端，HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响

应正文。

状态行的格式为： HTTP-Version Status-Code Reason-Phrase CRLF，其中HTTP-Version表示服务器HTTP协议的版本， Status-Code表示服务器返回的响应状态代码， Status-Code表示服务器返回的响应状态代码。

状态代码由三位数字组成，第一个数字定义了响应的类别，它有5种可能取值。

- （1）1xx： 指示信息。表示请求已接收，继续处理；
- （2）2xx： 成功。表示请求已被成功接收、理解、接受；
- （3）3xx： 重定向。要完成请求必须进行更进一步的操作；
- （4）4xx： 客户端错误。请求有语法错误或请求无法实现；
- （5）5xx： 服务器端错误。服务器未能处理请求。

常见的状态代码、状态描述如表10-2所示。

表10-2 HTTP响应状态代码和描述信息

响应报头允许服务器传递不能放在状态行中的附加响应信息， 以及关于服务器的信息和对Request-URI所标识的资源进行下一步访问的信息。常用的响应报头如表10-3所示。

表10-3 常用的响应报头

10.2 Netty HTTP服务端入门开发

从本节开始我们学习如何使用Netty的HTTP协议栈开发HTTP服务端和客户端应用程序。由于Netty天生是异步事件驱动的架构，因此基于NIO TCP协议栈开发的HTTP协议栈也是异步非阻塞的。

Netty的HTTP协议栈无论在性能还是可靠性上，都表现优异，非常适合在非Web容器的场景下应用，相比于传统的Tomcat、Jetty等Web容器，它更加轻量和小巧，灵活性和定制性也更好。

10.2.1 HTTP服务端例程场景描述

我们以文件服务器为例学习Netty的HTTP服务端入门开发，例程场景如下：文件服务器使用HTTP协议对外提供服务，当客户端通过浏览器访问文件服务器时，对访问路径进行检查，检查失败时返回HTTP 403错误，该页无法访问；如果校验通过，以链接的方式打开当前文件目录，每个目录或者文件都是个超链接，可以递归访问。

如果是目录，可以继续递归访问它下面的子目录或者文件，如果是文件且可读，则可以在浏览器端直接打开，或者通过【目标另存为】下载该文件。

介绍完了样例程序的开发场景，下面我们一起来看看如何开发一个基于Netty的HTTP程序。

10.2.2 HTTP服务端开发

首先看下HTTP文件服务器的启动类是如何实现的。

代码清单10-1 HTTP文件服务器 启动类 HttpFileServer

```
19. public class HttpFileServer {
20.
21.     private static final String DEFAULT_URL = "/src/com
22.
23.     public void run(final int port, final String url) {
24.         EventLoopGroup bossGroup = new NioEventLoopGroup();
25.         EventLoopGroup workerGroup = new NioEventLoopGroup(
26.             try {
27.                 ServerBootstrap b = new ServerBootstrap();
28.                 b.group(bossGroup, workerGroup)
29.                     .channel(NioServerSocketChannel.class)
30.                     .childHandler(new ChannelInitializer<Socket
31.                         @Override
32.                 protected void initChannel(SocketChannel ch
33.                     throws Exception {
34.                         ch.pipeline().addLast("http-decoder",
35.
36.
37.                         new HttpRequestDecoder());
38.
39.                         ch.pipeline().addLast("http-aggregator")
```

37. `new HttpObjectAggregator(65536));`

38. `ch.pipeline().addLast("http-encoder",`

39. `new HttpResponseEncoder());`

40. `ch.pipeline().addLast("http-chunked",`

41. `new ChunkedWriteHandler());`

42. `ch.pipeline().addLast("fileServerHandle`

```
43.                new HttpFileServerHandler(url));

44.                }
45.                });
46.                ChannelFuture future = b.bind("192.168.1.102",
47.                System.out.println("HTTP文件目录服务器启动, 网址是
48.                + port + url);
49.                future.channel().closeFuture().sync();
50.        } finally {
51.            bossGroup.shutdownGracefully();
52.            workerGroup.shutdownGracefully();
53.        }
54.    }
55.
56.    public static void main(String[] args) throws Excep
57.    int port = 8080;
58.    if (args.length > 0) {
59.        try {
60.            port = Integer.parseInt(args[0]);
61.        } catch (NumberFormatException e) {
62.            e.printStackTrace();
63.        }
64.    }
65.    String url = DEFAULT_URL;
66.    if (args.length > 1)
```



```
67.         url = args[1];
68.     new HttpFileServer().run(port, url);
69.     }
70. }
```

首先我们看main函数，它有两个参数：第一个是端口，第二个是HTTP服务端的URL路径。如果启动的时候没有配置，则使用默认值，默认端口是8080，默认的URL路径是“/src/com/phei/netty/”。

重点关注第34～43行，首先向ChannelPipeline中添加HTTP请求消息解码器，随后，又添加了HttpObjectAggregator解码器，它的作用是将多个消息转换为单一的FullHttpRequest或者FullHttpResponse，原因是HTTP解码器在每个HTTP消息中会生成多个消息对象。

(1) HttpRequest / HttpResponse;

(2) HttpContent;

(3) LastHttpContent。

第38～39行新增HTTP响应编码器，对HTTP响应消息进行编码；第40～41行新增Chunked handler，它的主要作用是支持异步发送大的码流（例如大的文件传输），但不占用过多的内存，防止发生Java内存溢出错误。

最后添加HttpFileServerHandler，用于文件服务器的业务逻辑处理。下面我们具体看看它是如何实现的。

代码清单10-2 HTTP文件服务器 处理类HttpFileServerHandler

```
47. public class HttpFileServerHandler extends
48.     SimpleChannelInboundHandler<FullHttpRequest> {
49.     private final String url;
50.
51.     public HttpFileServerHandler(String url) {
52.         this.url = url;
53.     }
54.
55.     @Override
56.     public void messageReceived(ChannelHandlerContext c
57.         FullHttpRequest request) throws Exception {
58.         if (!request.getDecoderResult().isSuccess()) {
59.             sendError(ctx, BAD_REQUEST);
60.             return;
61.         }
62.         if (request.getMethod() != GET) {
63.             sendError(ctx, METHOD_NOT_ALLOWED);
64.             return;
65.         }
66.         final String uri = request.getUri();
67.         final String path = sanitizeUri(uri);
68.         if (path == null) {
69.             sendError(ctx, FORBIDDEN);
70.             return;
71.         }
72.         File file = new File(path);
```

```
73.         if (file.isHidden() || !file.exists()) {
74.             sendError(ctx, NOT_FOUND);
75.             return;
76.         }
77.         if (file.isDirectory()) {
78.             if (uri.endsWith("/")) {
79.                 sendListing(ctx, file);
80.             } else {
81.                 sendRedirect(ctx, uri + '/');
82.             }
83.             return;
84.         }
85.         if (!file.isFile()) {
86.             sendError(ctx, FORBIDDEN);
87.             return;
88.         }
89.         RandomAccessFile randomAccessFile = null;
90.         try {
91.             randomAccessFile = new RandomAccessFile(file, "
92.         } catch (FileNotFoundException fnfe) {
93.             sendError(ctx, NOT_FOUND);
94.             return;
95.         }
96.         long fileLength = randomAccessFile.length();
97.         HttpResponse response = new DefaultHttpResponse(HTT
98.         setContentLength(response, fileLength);
99.         setContentTypeHeader(response, file);
```

```
100.         if (isKeepAlive(request)) {
101.             response.headers().set(CONNECTION, HttpHead
102.         }
103.         ctx.write(response);
104.         ChannelFuture sendFileFuture;
105.         sendFileFuture=ctx.write(new ChunkedFile(random
106.             fileLength, 8192), ctx.newProgressivePromis
107.         sendFileFuture.addListener(new ChannelProgressi
108.             @Override
109.             public void operationProgressed(ChannelProg
110.                 long progress, long total) {
111.                 if (total < 0) { // total unknown
112.                     System.err.println("Transfer progress:
113.                 } else {
114.                     System.err.println("Transfer progress:"
115.                         + total);
116.                 }
117.             }
118.
119.             @Override
120.             public void operationComplete(ChannelProgre
121.                 throws Exception {
122.                 System.out.println("Transfer complete.");
123.             }
124.         });
125.         ChannelFuture lastContentFuture = ctx
126.             .writeAndFlush>LastHttpContent.EMPTY_LAST_C
```

```
127.         if (!isKeepAlive(request)) {
128.             lastContentFuture.addListener(ChannelFuture
129.         }
130.     }
131.
132.     @Override
133.     public void exceptionCaught(ChannelHandlerContext
134.         throws Exception {
135.         cause.printStackTrace();
136.         if (ctx.channel().isActive()) {
137.             sendError(ctx, INTERNAL_SERVER_ERROR);
138.         }
139.     }
140.
141.     private static final Pattern INSECURE_URI = Pat
142.
143.     private String sanitizeUri(String uri) {
144.     try {
145.         uri = URLDecoder.decode(uri, "UTF-8");
146.     } catch (UnsupportedEncodingException e) {
147.         try {
148.             uri = URLDecoder.decode(uri, "ISO-8859-1");
149.         } catch (UnsupportedEncodingException e1) {
150.             throw new Error();
151.         }
152.     }
153.     if (!uri.startsWith(url)) {
```

```
154.         return null;
155.     }
156.     if (!uri.startsWith("/")) {
157.         return null;
158.     }
159.     uri = uri.replace('/', File.separatorChar);
160.     if (uri.contains(File.separator + '.')
161.         || uri.contains('.')+File.separator)||uri.st
162.         || uri.endsWith(".")||INSECURE_URI.matcher(
163.         return null;
164.     }
165.     return System.getProperty("user.dir") + File.se
166. }
167.
168.     private static final Pattern ALLOWED_FILE_NAME
169.         .compile("[A-Za-z0-9][_A-Za-z0-9\\.]*");
170.
171.     private static void sendListing(ChannelHandlerC
172. FullHttpResponse response = new DefaultFullHttp
173. response.headers().set(CONTENT_TYPE, "text/html;
174. StringBuilder buf = new StringBuilder();
175. String dirPath = dir.getPath();
176. buf.append("<!DOCTYPE html>\r\n");
177. buf.append("<html><head><title>");
178. buf.append(dirPath);
179. buf.append(" 目录: ");
180. buf.append("</title></head><body>\r\n");
```

```
181.         buf.append("<h3>");
182.         buf.append(dirPath).append(" 目录: ");
183.         buf.append("</h3>\r\n");
184.         buf.append("<ul>");
185.         buf.append("<li>链接: <a href=\"../\">../</a></li>");
186.         for (File f : dir.listFiles()) {
187.             if (f.isHidden() || !f.canRead()) {
188.                 continue;
189.             }
190.             String name = f.getName();
191.             if (!ALLOWED_FILE_NAME.matcher(name).matches)
192.                 continue;
193.             }
194.             buf.append("<li>链接: <a href=\"");
195.             buf.append(name);
196.             buf.append("\">");
197.             buf.append(name);
198.             buf.append("</a></li>\r\n");
199.         }
200.         buf.append("</ul></body></html>\r\n");
201.         ByteBuf buffer = Unpooled.copiedBuffer(buf, Char
202.             response.content().writeBytes(buffer);
203.             buffer.release();
204.             ctx.writeAndFlush(response).addListener (Channel
205.             }
206.
207.         private static void sendRedirect(ChannelHandler
```

```

208.         FullHttpResponse response = new DefaultFullHttp
209.         response.headers().set(LOCATION, newUri);
210.         ctx.writeAndFlush(response).addListener (Channe
211.         }
212.
213.         private static void sendError(ChannelHandlerCon
214.         HttpResponseStatus status) {
215.         FullHttpResponse response=new DefaultFullHttpRe
216.         status, Unpooled.copiedBuffer("Failure: " +
217.         + "\r\n", CharsetUtil.UTF_8));
218.         response.headers().set(CONTENT_TYPE, "text/plai
219.         ctx.writeAndFlush(response).addListener (Channe
220.         }
221.
222.         private static void setContentTyperHeader(HttpRe
223.         MimeTypesFileTypeMap mimeTypesMap=new MimeTypes
224.         response.headers().set(CONTENT_TYPE,
225.         mimeTypesMap.getContentType(file.getPath()))
226.         }
227.     }

```

首先从消息接入方法看起，第58～61行首先对HTTP请求消息的解码结果进行判断，

第67行对请求URL进行包装，然后对sanitizeUri方法展开分析。跳到第145行，首

第68～71行，如果构造的URI不合法，则返回HTTP 403错误。第72行使用新组装的

第172行首先创建成功的HTTP响应消息，随后设置消息头的类型为“text/html; c

如果用户在浏览器上点击超链接直接打开或者下载文件，代码会执行第85行，对超链

第96行获取文件的长度，构造成功的HTTP应答消息，然后在消息头中设置content

如果使用chunked编码，最后需要发送一个编码结束的空消息体，将LastHttpConi

如果是非Keep-Alive的，最后一包消息发送完成之后，服务端要主动关闭连接。

服务端的代码已经介绍完毕，下面让我们看看运行结果。

10.2.3 Netty HTTP文件服务器例程运行结果

启动HTTP文件服务器，通过浏览器进行访问，运行结果如下。

首先，启动文件服务器，运行结果如图10-1所示。

图10-1 HTTP文件服务器启动结果

我们首先进行异常场景的测试，输入错误的URL网址：

`http://192.168.1.102:8080/abcde/get?123`

运行结果如图10-2所示。

图10-2 输入错误的网址，返回403错误

结果分析：由于输入的URL路径不是个合法的文件或者目录，所以程序会执行第69行

我们继续测试正常场景，在浏览器中输入正确的网址：

`http://192.168.1.102:8080/src/com/phei/netty/`

浏览器显示结果如图10-3所示。

图10-3 Netty文件服务器目录展示

单击codec文件链接，显示结果如图10-4所示。

图10-4 单击目录链接，进入下一级目录

查看浏览器的网址，已经进入下一级目录，继续单击protobuf目录，显示如图10-5

图10-5 进入 protobuf目录

随便打开一个文件，由于是文本文件，可以直接在浏览器中显示，如图10-6所示。

图10-6 进入protobuf目录

如果内容显示有乱码，说明浏览器使用的编码方式和源代码中的不一致，直接在打开

图10-7 设置浏览器的编码方式为UTF-8解决中文乱码问题

我们也可以通过右键【目标另存为】从文件服务器下载文件，如图10-8所示。

图10-8 从文件服务器下载源文件

下载完成后，打开下载的文件，看内容是否正确，如图10-9所示。

图10-9 查看从文件服务器下载的源文件

对比服务器上的源文件，内容完全一致，说明HTTP文件服务器文件下载功能正常。

至此，作为入门级的Netty HTTP协议栈的应用——HTTP文件服务器已经开发完毕，本

下一节，我们将学习目前最流行的HTTP+XML开发。HTTP+XML应用非常广泛，一旦我

10.3 Netty HTTP+XML协议栈开发

由于HTTP协议的通用性，很多异构系统间的通信交互采用HTTP协议，通过HTTP协议

在Java领域，最常用的HTTP协议栈就是基于Servlet规范的Tomcat等Web容器，由

在网络安全日益严峻的今天，重量级的web容器由于功能繁杂，会存在很多安全漏洞，

本章节将介绍如何利用Netty提供的基础HTTP协议栈功能，扩展开发HTTP+XML协议

10.3.1 开发场景介绍

作为一个示例程序，我们先模拟一个简单的用户订购系统。客户端填写订单，通过HT

订购请求消息定义如表10-3所示。

表10-3 订购请求消息定义（Order）

客户信息定义如表10-4所示。

表10-4 客户信息定义（Customer）

地址信息定义如表10-5所示。

表10-5 地址信息定义（Address）

邮递方式定义如表10-6所示。

表10-6 邮递方式定义（Shipping）

数据定义完成之后，接着看订购请求消息的XML Schema定义。

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:
  <xs:element type="tns:order" name="order"/>
  <xs:complexType name="address">
    <xs:sequence>
      <xs:element type="xs:string" name="street1" minOccurs="
      <xs:element type="xs:string" name="street2" minOccurs="
      <xs:element type="xs:string" name="city" minOccurs="0"/
      <xs:element type="xs:string" name="state" minOccurs="0"
      <xs:element type="xs:string" name="postCode" minOccurs=
      <xs:element type="xs:string" name="country" minOccurs="
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="order">
    <xs:sequence>
      <xs:element name="customer" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string" name="firstName" min
            <xs:element type="xs:string" name="lastName" min0
            <xs:element type="xs:string" name="middleName" mi
          </xs:sequence>
          <xs:attribute type="xs:long" use="required" name="cus
        </xs:complexType>
      </xs:element>
      <xs:element type="tns:address" name="billTo" minOccurs=

```



```

<xs:element name="shipping" minOccurs="0">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="STANDARD_MAIL"/>
      <xs:enumeration value="PRIORITY_MAIL"/>
      <xs:enumeration value="INTERNATIONAL_MAIL"/>
      <xs:enumeration value="DOMESTIC_EXPRESS"/>
      <xs:enumeration value="INTERNATIONAL_EXPRESS"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element type="tns:address" name="shipTo" minOccurs=
</xs:sequence>

<xs:attribute type="xs:long" use="required" name="orderNu
<xs:attribute type="xs:float" name="total"/>
</xs:complexType></xs:schema>

```

熟悉XML和Schema的读者理解上面的Schema定义没有什么困难，如果你对XML的相

开发背景介绍完毕，下面我们进入设计环节，看看如何设计和开发HTTP+XML协议栈。

10.3.2 HTTP+XML协议栈设计

通过商品订购的流程图看下订购的关键步骤和主要技术点，找出当前Netty HTTP协

图10-10 HTTP+XML订购流程图

首先对订购流程图进行分析，先看步骤1，构造订购请求消息并将其编码为HTTP+XML

再看步骤2，利用Netty的HTTP协议栈，可以支持HTTP链路的建立和请求消息的发送

步骤3，HTTP服务端需要将HTTP+XML格式的订购请求消息解码为订购请求POJO对象

步骤4，服务端对订购请求消息处理完成后，重新将其封装成XML，通过HTTP应答消息

步骤5，HTTP客户端需要将HTTP+XML格式的应答消息解码为订购POJO对象，同时能

通过分析，我们可以了解到哪些能力是Netty支持的，哪些需要扩展开发实现。下面

(1) 需要一套通用、高性能的XML序列化框架，它能够灵活地实现POJO-XML的互相

(2) 作为通用的HTTP+XML协议栈，XML-POJO对象的映射关系应该非常灵活，支持

(3) 提供HTTP+XML请求消息编码器，供HTTP客户端发送请求消息自动编码使用；

(4) 提供HTTP+XML请求消息解码器，供HTTP服务端对请求消息自动解码使用；

(5) 提供HTTP+XML响应消息编码器，供HTTP服务端发送响应消息自动编码使用；

(6) 提供HTTP+XML响应消息解码器，供HTTP客户端对应答消息进行自动解码使用；

(7) 协议栈使用者不需要关心HTTP+XML的编解码，对上层业务零侵入，业务只需要

下个小节我们将讲述XML框架的选型和开发，它是HTTP+XML协议栈的关键技术点。

10.3.3 高效的XML绑定框架JiBX

JiBX是一款非常优秀的XML（Extensible Markup Language）数据绑定框架。’

1. JiBX入门

XML已经成为目前程序开发配置的重要组成部分了，可以用来操作XML文件的开源项目

使用JiBX绑定XML文档与Java对象需要分两步走：第一步是绑定XML文件，也就是明

在运行程序之前，需要先配置绑定文件并进行绑定，在绑定过程中它将会动态地修改

JiBX有两个比较重要的概念：Unmarshal（数据分解）和Marshal（数据编排）。

介绍完了JiBX的基础概念，下面我们就结合订购例程，来学习下如何使用JiBX进行)

2. POJO对象定义

通过JiBx提供的工具jar包，可以根据Schema自动生成POJO对象，也可以根据普通

考虑到大多数人的编码习惯，我们采用先定义POJO对象，再生成XML和对象的绑定文

代码清单10-3 HTTP+XML POJO类定义Order

```
2. public class Order {
3.     private long orderNumber;
4.     private Customer customer;
5.
6.     /** Billing address information. */
7.     private Address billTo;
8.
9.     private Shipping shipping;
10.
11.    /**
12.     * Shipping address information. If missing, the bill:
13.     * used as the shipping address.
14.     */
15.    private Address shipTo;
```

```
16.  
17.     private Float total;  
.....//定义set和get方法  
53. }
```

代码清单10-4 HTTP+XML POJO类定义Customer

```
2.  import java.util.List;  
3.  public class Customer {  
4.      private long customerNumber;  
5.      /** Personal name. */  
6.      private String firstName;  
7.  
8.      /** Family name. */  
9.      private String lastName;  
10.     /** Middle name(s), if any. */  
11.     private List<String> middleNames;  
.....//定义set和get方法  
35.     }
```

代码清单10-5 HTTP+XML POJO类定义Address

```
2.  public class Address {
3.      /** First line of street information (required). */
4.      private String street1;
5.      /** Second line of street information (optional). */
6.      private String street2;
7.      private String city;
8.
9.      /**
10.         * State abbreviation (required for the U.S. and Ca
11.         * otherwise).
12.         */
13.      private String state;
14.
15.      /** Postal code(required for the U.S.and Canada,opt
16.      private String postCode;
17.      /** Country name (optional, U.S. assumed if not sup
18.      private String country;
.....//定义set和get方法
54.      }
```

代码清单10-6 HTTP+XML POJO类定义Shipping

```
1. package com.phei.netty.protocol.http.xml.pojo;
2.
3. public enum Shipping {
4.     STANDARD_MAIL, PRIORITY_MAIL, INTERNATIONAL_MAIL, DO
5. }
```

POJO对象定义完成之后，通过Ant脚本来生成XML和POJO对象的绑定关系文件，同时

3. 通过Ant脚步生成XML和对象的绑定关系

首先确认你使用的Eclipse安装了Ant，通过【window】主菜单选择【Preferenc

图10-11 Eclipse的Ant配置

目前主流的Eclipse版本默认都支持Ant，如果你使用其他的开发工具，可以安装对

由于本书的重点并非讲解Ant的使用，所以，如果你对Ant感兴趣或者作为初学者学

JiBx的绑定和编译，重点关注两个task。如图10-12所示。

图10-12 使用BindGen命令生成绑定文件

通过JiBx的org.jibx.binding.generator.BindGen工具类可以将指定的POJO

图10-13 执行Ant脚本，生成XML绑定关系

执行成功后，在当前的工程目录下生成binding.xml和pojo.xsd文件，结果如图1

图10-14 生成的XML绑定文件

打开绑定文件，我们可以发现绑定文件实际就是XML的元素和POJO对象字段之间的映

图10-15 XML和Order对象的映射关系

再看一下JiBx的编译命令，它的作用是根据绑定文件和POJO对象的映射关系和规则：

图10-16 编译绑定脚本和动态修改Class的Ant脚本

编译结果如图10-17所示。

图10-17 动态修改Class文件

到此为止，JiBx相关的准备工作已经完成，下个小节通过一个简单的测试程序来学习

4. JiBx的类库使用

JiBx的类库使用非常简单，下面直接看代码。

代码清单10-7 HTTP+XML POJO测试类定义TestOrder

```
18. public class TestOrder {
19.     private IBindingFactory factory = null;
20.     private StringWriter writer = null;
21.     private StringReader reader = null;
22.     private final static String CHARSET_NAME = "UTF-8";
23.     private String encode2Xml(Order order) throws JiBXE
24.     factory = BindingDirectory.getFactory(Order.class);
25.     writer = new StringWriter();
26.     IMarshallingContext mctx = factory.createMarshalling
27.     mctx.setIndent(2);
28.     mctx.marshalDocument(order, CHARSET_NAME, null, wri
```

```
29.      String xmlStr = writer.toString();
30.      writer.close();
31.      System.out.println(xmlStr.toString());
32.      return xmlStr;
33.  }
34.
35.      private Order decode2Order(String xmlBody) throws J
36.      reader = new StringReader(xmlBody);
37.      IUnmarshallingContext uctx = factory.createUnmarsh
38.      Order order = (Order) uctx.unmarshalDocument(reader
39.      return order;
40.  }
41.
42.      public static void main(String[] args) throws JiBXE
43.      TestOrder test = new TestOrder();
44.      Order order = OrderFactory.create(123);
45.      String body = test.encode2Xml(order);
46.      Order order2 = test.decode2Order(body);
47.      System.out.println(order2);
48.  }
49. }
```

首先看第24行，根据Order的Class实例构造IBindingFactory对象。第25行创建

解码与编码类似，不同的是它使用StringReader来读取String类型的XML对象，

执行结果如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo" o
  <customer customerNumber="123">
    <firstName>李</firstName>
    <lastName>林峰</lastName>
  </customer>
  <billTo>
    <street1>龙眼大道</street1>
    <city>南京市</city>
    <state>江苏省</state>
    <postCode>123321</postCode>
    <country>中国</country>
  </billTo>
  <shipping>INTERNATIONAL_MAIL</shipping>
  <shipTo>
    <street1>龙眼大道</street1>
    <city>南京市</city>
```

```
<state>江苏省</state>
<postCode>123321</postCode>
<country>中国</country>
</shipTo>
</order>
Order [orderNumber=123, customer=Customer [customerNumber=123
```

通过上面的执行结果可以发现，XML序列化和反序列化后的结果与预期一致，我们开

XML绑定框架选型和开发完成之后，下面继续Netty HTTP +XM编解码框架的开发。

10.3.4 HTTP+XML编解码框架开发

本节共有6个小节来讲解如何基于Netty开发HTTP+XML协议栈，在Netty提供的HTT

1. HTTP+XML请求消息编码类

对于上层业务侧，构造订购请求消息后，以HTTP+XML协议将消息发送给服务端，如：

考虑到HTTP+XML协议栈需要一定的定制扩展能力，例如通过HTTP消息头携带业务自

HTTP+XML的协议编码仍然采用ChannelPipeline中增加对应的编码handler类实

下面我们来一起看下HTTP+XML请求消息编码类的源码实现。

代码清单10-8 HTTP+XML HTTP请求消息编码类

```
11. public class HttpXmlRequestEncoder extends
12.     AbstractHttpXmlEncoder<HttpXmlRequest> {
13.
14.     @Override
15.     protected void encode(ChannelHandlerContext ctx, Http
16.         List<Object> out) throws Exception {
17.         ByteBuf body = encode0(ctx, msg.getBody());
```

```
18.      FullHttpRequest request = msg.getRequest();

19.      if (request == null) {

20.          request = new DefaultFullHttpRequest(HttpVersio

21.              HttpMethod.GET, "/do", body);

22.          HttpHeaders headers = request.headers();

23.          headers.set(HttpHeaders.Names.HOST, InetAddress

24.              .getHostAddress());
```


25. `headers.set(HttpHeaders.Names.CONNECTION, HttpH`

26. `headers.set(HttpHeaders.Names.ACCEPT_ENCODING,`

27. `HttpHeaders.Values.GZIP.toString() + ', '`

28. `+ HttpHeaders.Values.DEFLATE.toString()`

29. `headers.set(HttpHeaders.Names.ACCEPT_CHARSET,`

30. `"ISO-8859-1, utf-8;q=0.7, *;q=0.7");`

31. `headers.set(HttpHeaders.Names.ACCEPT_LANGUAGE,`

32. *headers.set(HttpHeaders.Names.USER_AGENT,*

33. *"Netty xml Http Client side");*

34. *headers.set(HttpHeaders.Names.ACCEPT,*

35. *}*

36. *HttpHeaders.setContentLength(request, body.readable*

37. *out.add(request);*

38. *}*

39. }

第17行首先调用父类的`encode0`，将业务需要发送的POJO对象`Order`实例通过`JiBx`

第20~35行用来构造和设置默认的HTTP消息头，由于通常情况下HTTP通信双方更关

第36行很重要，由于请求消息消息体不为空，也没有使用`Chunk`方式，所以在HTTP

下面我们来看父类`AbstractHttpXmlEncoder`的实现。

代码清单10-9 HTTP+XML HTTP请求消息编码基类 `AbstractHttp`

```
14. public abstract class AbstractHttpXmlEncoder<T> extends
15.     MessageToMessageEncoder<T> {
16.     IBindingFactory factory = null;
17.     StringWriter writer = null;
18.     final static String CHARSET_NAME = "UTF-8";
```

```
19.         final static Charset UTF_8 = Charset.forName(CHARSE
20.
21.         protected ByteBuf encode0(ChannelHandlerContext ctx
22.             throws Exception {
23.             factory = BindingDirectory.getFactory(body.getClass

24.             writer = new StringWriter();

25.             IMarshallingContext mctx = factory.createMarshallin

26.             mctx.setIndent(2);

27.             mctx.marshalDocument(body, CHARSET_NAME, null, writ

28.             String xmlStr = writer.toString();
```

```
29.         writer.close();

30.         writer = null;

31.         ByteBuf encodeBuf = Unpooled.copiedBuffer(xmlStr, U

32.         return encodeBuf;

33.     }
34.
35.     @Override
36.     public void exceptionCaught(ChannelHandlerContext c
37.         throws Exception {
38.         // 释放资源
39.         if (writer != null) {
40.             writer.close();
41.             writer = null;
42.         }
```

```
43.      }
44.  }
```

首先看下23~30行，代码很熟悉，在JiBx章节已经介绍了XML序列化和反序列化的木

下面继续看下HttpXmlRequest是如何实现的。

代码清单10-10 HTTP+XML请求消息HttpXmlRequest

```
9.      public class HttpXmlRequest {
10.          private FullHttpRequest request;
11.          private Object body;
12.
13.          public HttpXmlRequest(FullHttpRequest request, Obje
14.              this.request = request;
15.              this.body = body;
16.          }
17.
18.          /**
```

```
19.      * @return the request
20.      */
21.      public final FullHttpRequest getRequest() {
22.          return request;
23.      }
24.
25.      /**
26.       * @param request
27.       *           the request to set
28.       */
29.      public final void setRequest(FullHttpRequest request) {
30.          this.request = request;
31.      }
32.
33.      /**
34.       * @return the object
35.       */
36.      public final Object getBody() {
37.          return body;
38.      }
39.
40.      /**
41.       * @param object
42.       *           the object to set
43.       */
44.      public final void setBody(Object body) {
45.          this.body = body;
```

```
46.    }
```

```
47.    }
```

它包含两个成员变量`FullHttpRequest`和编码对象`Object`，用于实现和协议栈之间的

2. HTTP+XML请求消息解码类

HTTP服务端接收到HTTP+XML请求消息后，需要从HTTP消息体中获取请求码流，通过

下面就看下具体实现。

代码清单10-11 HTTP+XML HTTP请求消息解码类 `HttpXmlRequest`

```
20.    public class HttpXmlRequestDecoder extends
21.        AbstractHttpXmlDecoder<FullHttpRequest> {
22.
23.        public HttpXmlRequestDecoder(Class<?> clazz) {
```



```
24.         this(clazz, false);
25.     }
26.
27.     public HttpXmlRequestDecoder(Class<?> clazz, boolean isPrint) {
28.         super(clazz, isPrint);
29.     }
30.
31.     @Override
32.     protected void decode(ChannelHandlerContext arg0, FullHttpRequest arg1,
33.         List<Object> arg2) throws Exception {
34.         if (!arg1.getDecoderResult().isSuccess()) {
35.             sendError(arg0, BAD_REQUEST);
36.             return;
37.         }
38.         HttpXmlRequest request = new HttpXmlRequest(arg1, d
```

```
39.         arg1.content()));

40.     arg2.add(request);

41. }
42.
43.     private static void sendError(ChannelHandlerContext
44.         HttpResponseStatus status) {
45.         FullHttpResponse response = new DefaultFullHttpResp

46.         status, Unpooled.copiedBuffer("Failure: " + sta

47.         + "\r\n", CharsetUtil.UTF_8));
```

```
48.         response.headers().set(CONTENT_TYPE, "text/plain; c

49.         ctx.writeAndFlush(response).addListener(ChannelFutu

50.     }

51. }
```

`HttpXmlRequestDecoder`有两个参数，分别为需要解码的对象的类型信息和是否打

第43~50行，如果HTTP消息本身解码失败，则构造处理结果异常的HTTP应答消息返

第38行通过`HttpXmlRequest`和反序列化后的`Order`对象构造`HttpXmlRequest`实

继续看下它的父类`AbstractHttpXmlDecoder`的实现。

代码清单10-12 HTTP+XML HTTP请求消息解码类 `AbstractHttpX`

```
18. public abstract class AbstractHttpXmlDecoder<T> extends
19.     MessageToMessageDecoder<T> {
20.     private IBindingFactory factory;
21.     private StringReader reader;
22.     private Class<?> clazz;
23.     private boolean isPrint;
24.     private final static String CHARSET_NAME = "UTF-8";
25.     private final static Charset UTF_8 = Charset.forName(CHARSET_NAME);
26.
27.     protected AbstractHttpXmlDecoder(Class<?> clazz) {
28.         this(clazz, false);
29.     }
30.
31.     protected AbstractHttpXmlDecoder(Class<?> clazz, boolean isPrint) {
32.         this.clazz = clazz;
33.         this.isPrint = isPrint;
34.     }
35.
36.     protected Object decode0(ChannelHandlerContext arg0,
37.                             ByteBuffer arg1,
38.                             throws Exception {
39.         factory = BindingDirectory.getFactory(clazz);
```

```
39.      String content = body.toString(UTF_8);

40.      if (isPrint)

41.          System.out.println("The body is : " + content);

42.      reader = new StringReader(content);

43.      IUnmarshallingContext uctx = factory.createUnmarsha

44.      Object result = uctx.unmarshalDocument(reader);

45.      reader.close();
```

```
46.         reader = null;

47.         return result;

48.     }
49.     @Override
50.     public void exceptionCaught(ChannelHandlerContext c
51.         throws Exception {
52.         // 释放资源
53.         if (reader != null) {
54.             reader.close();
55.             reader = null;
56.         }
57.     }
58. }
```

第38~47行从HTTP的消息体中获取请求码流，然后通过JiBx类库将XML转换成POJO

第53~55行，如果解码发生异常，要判断StringReader是否已经关闭，如果没有关

3. HTTP+XML响应消息编码类

对于响应消息，用户可能并不关心HTTP消息头之类的，它将业务处理后的POJO对象

代码清单10-13 HTTP+XML HTTP XML应答消息 HttpXmlRespor

```
9.     public class HttpXmlResponse {
10.         private FullHttpResponse httpResponse;
11.         private Object result;
12.
13.         public HttpXmlResponse(FullHttpResponse httpResponse) {
14.             this.httpResponse = httpResponse;
15.             this.result = result;
16.         }
17.
18.         /**
19.          * @return the httpResponse
```

```
20.         */
21.     public final FullHttpResponse getHttpResponse() {
22.         return httpResponse;
23.     }
24.
25.     /**
26.      * @param httpResponse
27.      *         the httpResponse to set
28.      */
29.     public final void setHttpResponse(FullHttpResponse
30.         httpResponse) {
31.         this.httpResponse = httpResponse;
32.     }
33.
34.     /**
35.      * @return the body
36.      */
37.     public final Object getResult() {
38.         return result;
39.     }
40.
41.     /**
42.      * @param body
43.      *         the body to set
44.      */
45.     public final void setResult(Object result) {
46.         this.result = result;
47.     }
```



```
47.    }
```

它包含两个成员变量：`FullHttpResponse`和`Object`，`Object`就是业务需要发送的

下面继续看应答消息的XML编码类实现。

代码清单10-14 HTTP+XML 应答消息编码类 `HttpXmlResponseEr`

```
17.    public class HttpXmlResponseEncoder extends
18.        AbstractHttpXmlEncoder<HttpXmlResponse> {
19.
20.        /*
21.         * (non-Javadoc)
22.         *
23.         * @see
24.         * io.netty.handler.codec.MessageToMessageEncoder#e
25.         * .ChannelHandlerContext, java.lang.Object, java.u
26.         */
27.        protected void encode(ChannelHandlerContext ctx, Http
```

```
28.         List<Object> out) throws Exception {
29.     ByteBuf body = encode0(ctx, msg.getResult());
30.     FullHttpResponse response = msg.getHttpResponse();
31.     if (response == null) {
32.         response = new DefaultFullHttpResponse(HTTP_1_1
33.     } else {
34.         response = new DefaultFullHttpResponse(msg.getH
35.             .getProtocolVersion(), msg.getHttpResponse(
36.                 body);
37.     }
38.     response.headers().set(CONTENT_TYPE, "text/xml");
39.     setContentLength(response, body.readableBytes());
40.     out.add(response);
41. }
42. }
```

它的实现非常简单，第30行对应答消息进行判断，如果业务侧已经构造了HTTP应答消

作为示例程序并没有提供更多的API供业务侧灵活设置HTTP应答消息头，在实际商用

第38行设置消息体内容格式为“text/xml”，然后在消息头中设置消息体的长度。第

4. HTTP+XML应答消息解码

客户端接收到HTTP+XML应答消息后，对消息进行解码，获取HttpXmlResponse对象

代码清单10-15 HTTP+XML 应答消息解码类 HttpXmlResponseDecoder

```
11. public class HttpXmlResponseDecoder extends
12.     AbstractHttpXmlDecoder<DefaultFullHttpResponse> {
13.
14.     public HttpXmlResponseDecoder(Class<?> clazz) {
15.         this(clazz, false);
16.     }
17.
18.     public HttpXmlResponseDecoder(Class<?> clazz, boolean
19.         super(clazz, isPrintlog);
20.     }
21.
22.     @Override
23.     protected void decode(ChannelHandlerContext ctx,
24.         DefaultFullHttpResponse msg, List<Object> out)
```

```
25.         HttpXmlResponse resHttpXmlResponse = new HttpXmlRes
26.             ctx, msg.content());
27.         out.add(resHttpXmlResponse);
28.     }
29. }
```

第25行通过DefaultFullHttpResponse和HTTP应答消息反序列化后的POJO对象

5. HTTP+XML客户端开发

客户端的功能如下。

(1) 发起HTTP连接请求；

(2) 构造订购请求消息，将其编码成XML，通过HTTP协议发送给服务端；

(3) 接收HTTP服务端的应答消息，将XML应答消息反序列化为订购消息POJO对象；

(4) 关闭HTTP连接。

基于它的功能定位，我们首先开始主程序的开发。

代码清单10-16 HTTP+XML客户端启动类 HttpXmlClient

[illegible]

```
37.         new HttpResponseDecoder());

38.         ch.pipeline().addLast("http-aggregator"

39.         new HttpObjectAggregator(65536));

40.         // XML 解码器

41.         ch.pipeline().addLast(

42.         "xml-decoder",

43.         new HttpXmlResponseDecoder(Order.cl
```

44. `true));`

45. `ch.pipeline().addLast("http-encoder",`

46. `new HttpRequestEncoder());`

47. `ch.pipeline().addLast("xml-encoder",`

48. `new HttpXmlRequestEncoder());`

49. `ch.pipeline().addLast("xmlClientHandler`

50. `new HttpXmlClientHandle());`

```

51.         }
52.     });
53.
54.     // 发起异步连接操作
55.     ChannelFuture f=b.connect(new InetSocketAddress
56.
57.     // 等待客户端链路关闭
58.     f.channel().closeFuture().sync();
59. } finally {
60.     // 优雅退出，释放NIO线程组
61.     group.shutdownGracefully();
62. }
63. }
64.
65. /**
66.  * @param args
67.  * @throws Exception
68.  */
69. public static void main(String[] args) throws Excep
70. int port = 8080;
71. if (args != null && args.length > 0) {
72.     try {
73.         port = Integer.valueOf(args[0]);
74.     } catch (NumberFormatException e) {

```



```
75.          // 采用默认值
76.      }
77.  }
78.      new HttpXmlClient().connect(port);
79.  }
80. }
```

在ChannelPipeline中新增了HttpResponseDecoder，它负责将二进制码流解码

第45~46行将HttpRequestEncoder编码器添加到ChannelPipeline中时，需要

最后是业务的逻辑编排类HttpXmlClientHandle，我们继续分析它的实现。

代码清单10-17 HTTP+XML客户端业务逻辑编排类 HttpXmlClien

```
7.      public class HttpXmlClientHandle extends
8.          SimpleChannelInboundHandler<HttpXmlResponse> {
9.
```

```

10.      @Override
11.      public void channelActive(ChannelHandlerContext ctx
12.      HttpRequest request = new HttpRequest(null,
13.          OrderFactory.create(123));
14.      ctx.writeAndFlush(request);
15.  }
16.
17.      @Override
18.      public void exceptionCaught(ChannelHandlerContext c
19.      cause.printStackTrace();
20.      ctx.close();
21.  }
22.
23.      @Override
24.      protected void messageReceived(ChannelHandlerContext
25.          HttpRequestResponse msg) throws Exception {
26.      System.out.println("The client receive response of
27.          + msg.getHttpResponse().headers().names());
28.      System.out.println("The client receive response of
29.          + msg.getResult());
30.  }
31.  }

```

客户端的实现非常简单，第12行构造HttpRequest对象，调用ChannelHandle

第24~30行用于接收服务端的应答消息，从接口看，它接收到的已经是自动解码后的

最后，看下订购对象工厂类的实现。

代码清单10-18 HTTP+XML订购对象工厂类 OrderFactory

```
2.    public class OrderFactory {
3.        public static Order create(long orderID) {
4.            Order order = new Order();
5.            order.setOrderNumber(orderID);
6.            order.setTotal(9999.999f);
7.            Address address = new Address();
8.            address.setCity("南京市");
9.            address.setCountry("中国");
10.           address.setPostCode("123321");
11.           address.setState("江苏省");
12.           address.setStreet1("龙眠大道");
13.           order.setBillTo(address);
14.           Customer customer = new Customer();
15.           customer.setCustomerNumber(orderID);
```

```
16.         customer.setFirstName("李");
17.         customer.setLastName("林峰");
18.         order.setCustomer(customer);
19.         order.setShipping(Shipping.INTERNATIONAL_MAIL);
20.         order.setShipTo(address);
21.         return order;
22.     }
23. }
```

6. HTTP+XML服务端开发

HTTP服务端的功能如下。

- (1) 接收HTTP客户端的连接；
- (2) 接收HTTP客户端的XML请求消息，并将其解码为POJO对象；
- (3) 对POJO对象进行业务处理，构造应答消息返回；

(5) 主动关闭HTTP连接。

代码清单10-19 HTTP+XML服务端主程序HttpXmlServer

[illegible]

35. `new HttpRequestDecoder());`

36. `ch.pipeline().addLast("http-aggregator"`

37. `new HttpObjectAggregator(65536));`

38. `ch.pipeline()`

39. `.addLast(`

40. `"xml-decoder",`

41. `new HttpXmlRequestDecoder(`

42. `Order.class, true));`

43. `ch.pipeline().addLast("http-encoder",`

44. `new HttpResponseEncoder());`

45. `ch.pipeline().addLast("xml-encoder",`

46. `new HttpXmlResponseEncoder());`

47. `ch.pipeline().addLast("xmlServerHandler`

```
48.                new HttpXmlServerHandler());

49.                }
50.                });
51.                ChannelFuture future = b.bind(new InetSocketAddress
52.                System.out.println("HTTP订购服务器启动, 网址是 : "
53.                + port);
54.                future.channel().closeFuture().sync();
55.        } finally {
56.            bossGroup.shutdownGracefully();
57.            workerGroup.shutdownGracefully();
58.        }
59.    }
60.
61.    public static void main(String[] args) throws Excep
62.    int port = 8080;
63.    if (args.length > 0) {
64.        try {
65.            port = Integer.parseInt(args[0]);
66.        } catch (NumberFormatException e) {
67.            e.printStackTrace();
68.        }
69.    }
70.    new HttpXmlServer().run(port);
```



```
71.         }
72.     }
```

HTTP服务端的启动与之前一样，在此不再详述，我们具体看下编解码handler是如何

第34~37行用于绑定HTTP请求消息解码器；第38~42行将我们自定义的HttpXml

下面我们继续看HttpXmlServerHandler的实现。

代码清单10-20 HTTP+XML 服务端处理类 HttpXmlServerHandler

```
30.     public class HttpXmlServerHandler extends
31.         SimpleChannelInboundHandler<HttpXmlRequest> {
32.
33.         @Override
34.         public void messageReceived(final ChannelHandlerContext
35.             HttpXmlRequest xmlRequest) throws Exception {
36.             HttpRequest request = xmlRequest.getRequest();
```

```
37.      Order order = (Order) xmlRequest.getBody();

38.      System.out.println("Http server receive request : "

39.      dobusiness(order);

40.      ChannelFuture future = ctx.writeAndFlush(new HttpXm

41.      order));

42.      if (!isKeepAlive(request)) {
```

43. *future.addListener(new GenericFutureListener<Fu*

44. *public void operationComplete(Future future) th*

45. *ctx.close();*

46. *}*

47. *});*

48. *}*

49. *}*

50.

51. *private void dobusiness(Order order) {*

52. *order.getCustomer().setFirstName("狄");*

53. *order.getCustomer().setLastName("仁杰");*

54. *List<String> midNames = new ArrayList<String>();*

```
55.         midNames.add("李元芳");
56.         order.getCustomer().setMiddleNames(midNames);
57.         Address address = order.getBillTo();
58.         address.setCity("洛阳");
59.         address.setCountry("大唐");
60.         address.setState("河南道");
61.         address.setPostCode("123456");
62.         order.setBillTo(address);
63.         order.setShipTo(address);
64.     }
65.
66.     @Override
67.     public void exceptionCaught(ChannelHandlerContext c
68.         throws Exception {
69.         cause.printStackTrace();
70.         if (ctx.channel().isActive()) {
71.             sendError(ctx, INTERNAL_SERVER_ERROR);
72.         }
73.     }
74.
75.     private static void sendError(ChannelHandlerContext
76.         HttpResponseStatus status) {
77.         FullHttpResponse response = new DefaultFullHttpResp
78.             status, Unpooled.copiedBuffer("失败: " + status.
79.                 + "\r\n", CharsetUtil.UTF_8));
80.         response.headers().set(CONTENT_TYPE,"text/plain; ch
81.         ctx.writeAndFlush(response).addListener(ChannelFutu
```

```
82.      }  
83.  }
```

通过messageReceived的方法入参HttpXmlRequest，可以看出服务端业务处理类

第70~71行，在发生异常并且链路没有关闭的情况下，构造内部异常消息发送给客户

到此，HTTP+XML的协议栈开发工作全部完成，下个小节我们看下运行结果。

10.3.5 HTTP+XML协议栈测试

本小节对前面几节开发的HTTP+XML协议栈进行测试。

首先对工程进行编译，然后执行JiBx的Ant脚本，对涉及的POJO对象进行二次编译，

1. 服务端

服务端接收到的请求消息码流打印如下。

```
The body is : <?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo" o
  <customer customerNumber="123">
    <firstName>李</firstName>
    <lastName>林峰</lastName>
  </customer>
  <billTo>
    <street1>龙眼大道</street1>
    <city>南京市</city>
    <state>江苏省</state>
    <postCode>123321</postCode>
    <country>中国</country>
  </billTo>
  <shipping>INTERNATIONAL_MAIL</shipping>
  <shipTo>
    <street1>龙眼大道</street1>
    <city>南京市</city>
    <state>江苏省</state>
    <postCode>123321</postCode>
    <country>中国</country>
```

```
</shipTo>
</order>
```

服务端解码后的业务对象如下。

Http server receive request : Order [orderNumber=123, custome

2. 客户端

客户端接收到的响应消息体码流如下。

```
The body is : <?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo" o
  <customer customerNumber="123">
    <firstName>狄</firstName>
```

```
<lastName>仁杰</lastName>
  <middleName>李元芳</middleName>
</customer>
<billTo>
  <street1>龙眼大道</street1>
  <city>洛阳</city>
  <state>河南道</state>
  <postCode>123456</postCode>
  <country>大唐</country>
</billTo>
<shipping>INTERNATIONAL_MAIL</shipping>
<shipTo>
  <street1>龙眼大道</street1>
  <city>洛阳</city>
  <state>河南道</state>
  <postCode>123456</postCode>
  <country>大唐</country>
</shipTo>
</order>
```

解码后的响应消息如下。

The client receive response of http body is : Order [orderNum

测试结果表明，HTTP+XML协议栈功能正常，达到了设计预期。

10.3.6 小结

需要指出的是，尽管本章节开发的HTTP+XML协议栈是个高性能、通用的协议栈，但

10.4 总结

本章节重点介绍了HTTP协议以及如何使用Netty的HTTP协议栈开发基于HTTP的应用

本章节的HTTP+XML协议栈在实际项目中非常有用，如果读者打算以它为基础进行商

第11章 WebSocket协议开发

一直以来，网络在很大程度上都是围绕着HTTP的请求/响应模式而构建的。客户端加载一个网页，然后直到用户点击下一页之前，什么都不会发生。在2005年左右，Ajax开始让网络变得更加动态了。但所有的HTTP通信仍然是由客户端控制的，这就需要用户进行互动或定期轮询，以便从服务器加载新数据。

长期以来存在着各种技术让服务器得知有新数据可用时，立即将数据发送到客户端。这些技术种类繁多，例如“推送”或Comet。最常用的一种黑客手段是对服务器发起链接创建假象，被称为长轮询。利用长轮询，客户端可以打开指向服务器的HTTP连接，而服务器会一直保持连接打开，直到发送响应。服务器只要实际拥有新数据，就会发送响应（其他技术包括Flash、XHR multipart请求和所谓的HTML Files）。长轮询和其他技术都非常好用，在Gmail聊天等应用中会经常使用它们。

但是，这些解决方案都存在一个共同的问题：由于HTTP协议的开销，导致它们不适用于低延迟应用。

为了解决这些问题，WebSocket将网络套接字引入到了客户端和服务端，浏览器和服务器之间可以通过套接字建立持久的连接，双方随时都可以互发数据给对方，而不是之前由客户端控制的一请求一应答模式。

本章主要包括：

- HTTP协议的弊端

- WebSocket入门
- Netty WebSocket协议开发

11.1 HTTP协议的弊端

将HTTP协议的主要弊端总结如下。

(1) HTTP协议为半双工协议。半双工协议指数据可以在客户端和服务端两个方向上传输，但是不能同时传输。它意味着在同一时刻，只有一个方向上的数据传送；

(2) HTTP消息冗长而繁琐。HTTP消息包含消息头、消息体、换行符等，通常情况下采用文本方式传输，相比于其他的二进制通信协议，冗长而繁琐；

(3) 针对服务器推送的黑客攻击。例如长时间轮询。

现在，很多网站为了实现消息推送，所用的技术都是轮询。轮询是在特定的时间间隔（如每1秒），由浏览器对服务器发出HTTP request，然后由服务器返回最新的数据给客户端浏览器。这种传统的模式具有很明显的缺点，即浏览器需要不断地向服务器发出请求，然而HTTP request的header是非常冗长的，里面包含的可用数据比例可能非常低，这会占用很多的带宽和服务器资源。

比较新的一种轮询技术是Comet，使用了Ajax。这种技术虽然可达到双向通信，但依然需要发出请求，而且在Comet中，普遍采用了长连接，这也会大量消耗服务器带宽和资源。

为了解决HTTP协议效率低下的问题，HTML5定义了WebSocket协议，能更好地节省服务器资源和带宽并达到实时通信，下个小节让我们一起来学习WebSocket的入门知识。

11.2 WebSocket入门

WebSocket是HTML5开始提供的一种浏览器与服务器间进行全双工通信的网络技术，WebSocket通信协议于2011年被IETF定为标准RFC6455，WebSocket API被W3C定为标准。

在WebSocket API中，浏览器和服务器只需要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道，两者就可以直接互相传送数据了。WebSocket基于TCP双向全双工进行消息传递，在同一时刻，既可以发送消息，也可以接收消息，相比于HTTP的半双工协议，性能得到很大提升。

下面总结一下WebSocket的特点。

- 单一的TCP连接，采用全双工模式通信；
- 对代理、防火墙和路由器透明；
- 无头部信息、Cookie和身份验证；
- 无安全开销；
- 通过“ping/pong”帧保持链路激活；
- 服务器可以主动传递消息给客户端，不再需要客户端轮询。

11.2.1 WebSocket背景

WebSocket 设计出来的目的就是要取代轮询和Comet技术，使客户端浏览器具备像C/S架构下桌面系统一样的实时通信能力。浏览器通过JavaScript向服务器发出建立WebSocket连接的请求，连接建立以后，客户端和服务端可以通过TCP连接直接交换数据。因为WebSocket连接本质上就是一个TCP连接，所以在数据传输的稳定性和数据传输量的大

小方面，和轮询以及Comet技术相比，具有很大的性能优势。Websocket.org网站对传统的轮询方式和WebSocket调用方式作了一个详细的测试和比较，将一个简单的Web应用分别通过轮询方式和WebSocket方式来实现，在这里引用一下测试结果，如图11-1所示。

图11-1 轮询和WebSocket网络负载对比图

通过对比图可以清楚地看出，在流量和负载增大的情况下，WebSocket方案相比传统的Ajax轮询方案有极大的性能优势。这也是为什么我们认为WebSocket是未来实时Web应用的首选方案的原因。

11.2.2 WebSocket连接建立

客户端和服务端连接建立的示意图如图11-2所示。

图11-2 客户端和服务端握手连接

建立WebSocket连接时，需要通过客户端或者浏览器发出握手请求，请求消息示例如图11-3所示。

图11-3 WebSocket客户端握手请求消息

为了建立一个WebSocket连接，客户端浏览器首先要向服务器发起一个HTTP请求，这个请求和通常的HTTP请求不同，包含了一些附加头信息，其中附加头信息“Upgrade: WebSocket”表明这是一个申请协议升级的HTTP请求。服务器端解析这些附加的头信息，然后生成应答信息返回给客户端，客户端和服务端端的WebSocket连接就建立起来了，双方可以通过这个连接通道自由地传递信息，并且这个连接会持续存在直到客户端或者服务器端的某一方主动关闭连接。

服务端返回给客户端的应答消息如图11-4所示。

图11-4 WebSocket服务端返回的握手应答消息

请求消息中的“Sec-WebSocket-Key”是随机的，服务器端会用这些数据来构造出一个SHA-1的信息摘要，把“Sec-WebSocket-Key”加上一个魔幻字符串“258EAF5E-E914-47DA-95CA-C5AB0DC85B11”。使用SHA-1加密，然后进行BASE-64编码，将结果做为“Sec-WebSocket-Accept”头的值，返回给客户端。

11.2.3 WebSocket生命周期

握手成功之后，服务端和客户端就可以通过“messages”的方式进行通信了，一个消息由一个或者多个帧组成，WebSocket的消息并不一定对应一个特定网络层的帧，它可以被分割成多个帧或者被合并。

帧都有自己对应的类型，属于同一个消息的多个帧具有相同类型的数据。从广义上讲，数据类型可以是文本数据（UTF-8[RFC3629]文字）、二进制数据和控制帧（协议级信令，如信号）。

WebSocket连接生命周期示意图如图11-5所示。

图11-5 WebSocket生命周期示意图

11.2.4 WebSocket连接关闭

为关闭WebSocket连接，客户端和服务端需要通过一个安全的方法关闭底层TCP连接以及TLS会话。如果合适，丢弃任何可能已经接收的字节；必要时（比如受到攻击），可以通过任何可用的手段关闭连接。

底层的TCP连接，在正常情况下，应该首先由服务器关闭。在异常情况下（例如在一个合理的时间周期后没有接收到服务器的TCP Close），客户端可以发起TCP Close。因此，当服务器被指示关闭WebSocket连接时，它应该立即发起一个TCP Close操作；客户端应该等待服务器的TCP Close。

WebSocket的握手关闭消息带有一个状态码和一个可选的关闭原因，它必须按照协议要求发送一个Close控制帧，当对端接收到关闭控制帧指令时，需要主动关闭WebSocket连接。

通过本节的描述，相信读者对WebSocket的基础知识有了一定的了解，大家如果对WebSocket规范感兴趣，可以访问WebSocket的官网去了解更多的相关知识。

下个小节我们将一起学习如何使用Netty开发WebSocket服务端。

11.3 Netty WebSocket协议开发

Netty基于HTTP协议栈开发了WebSocket协议栈，利用Netty的WebSocket协议栈可以非常方便地开发出WebSocket客户端和服务端。本节通过一个Netty服务端实例的开发，向读者讲解如何使用Netty进行WebSocket开发。

11.3.1 WebSocket服务端功能介绍

WebSocket服务端的功能如下：支持WebSocket的浏览器通过WebSocket协议发送请求消息给服务端，服务端对请求消息进行判断，如果是合法的WebSocket请求，则获取请求消息体（文本），并在后面追加字符串“欢迎使用Netty WebSocket服务，现在时刻：系统时间”。

客户端HTML通过内嵌的JS脚本创建WebSocket连接，如果握手成功，在文本框中打印“打开WebSocket服务正常，浏览器支持WebSocket!”。客户端界面如图11-6所示。

图11-6 WebSocket客户端HTML

当前支持WebSocket的浏览器如图11-7所示。从图中可以看出，目前主流的浏览器都已经支持WebSocket，但是在运行本例程之前你仍然需要确认自己使用的浏览器版本是否已经支持WebSocket，否则会提示“抱歉，您的浏览器不支持WebSocket协议!”。

图11-7 支持WebSocket的浏览器及其版本

11.3.2 WebSocket服务端开发

首先对WebSocket服务端的功能进行简单地讲解。WebSocket服务端接收到请求消息之后，先对消息的类型进行判断，如果不是WebSocket握手请求消息，则返回 HTTP 400 BAD REQUEST 响应给客户端。客户端的握手请求消息如图11-8所示。

图11-8 客户端发送的WebSocket握手请求消息

服务端对握手请求消息进行处理，构造握手响应返回，双方的Socket连接正式建立，服务端返回的握手应答消息如图11-9所示。

图11-9 WebSocket握手应答消息

连接建立成功后，到被关闭之前，双方都可以主动向对方发送消息，这点跟HTTP的一请求一应答模式存在很大的差别。相比于HTTP，它的网络利用率更高，可以通过全双工的方式进行消息发送和接收。

下面一起来看下服务端代码的具体实现，首先看服务启动类。

代码清单11-1 WebSocket服务端启动类 WebSocketServer

```
1.    public class WebSocketServer {
2.        public void run(int port) throws Exception {
3.            EventLoopGroup bossGroup = new NioEventLoopGroup();
4.            EventLoopGroup workerGroup = new NioEventLoopGroup(
5.                try {
6.                    ServerBootstrap b = new ServerBootstrap();
7.                    b.group(bossGroup, workerGroup)
8.                        .channel(NioServerSocketChannel.class)
9.                        .childHandler(new ChannelInitializer<Socket
```

```
10.
11.         @Override
12.         protected void initChannel(SocketChannel ch
13.             throws Exception {
14.             ChannelPipeline pipeline = ch.pipeline(
15.                 pipeline.addLast("http-codec",

16.                     new HttpServerCodec());

17.                 pipeline.addLast("aggregator",

18.                     new HttpObjectAggregator(65536));

19.                 ch.pipeline().addLast("http-chunked",

20.                     new ChunkedWriteHandler());
```

```
21.                pipeline.addLast("handler",

22.                new WebSocketServerHandler());

23.                }
24.                });
25.
26.                Channel ch = b.bind(port).sync().channel();
27.                System.out.println("Web socket server started a
28.                + '.');
29.                System.out
30.                .println("Open your browser and navigate to
31.                + port + '/');
32.
33.                ch.closeFuture().sync();
34.        } finally {
35.                bossGroup.shutdownGracefully();
36.                workerGroup.shutdownGracefully();
37.        }
38.        }
39.
```

```
40.         public static void main(String[] args) throws Excep
41.         int port = 8080;
42.         if (args.length > 0) {
43.             try {
44.                 port = Integer.parseInt(args[0]);
45.             } catch (NumberFormatException e) {
46.                 e.printStackTrace();
47.             }
48.         }
49.         new WebSocketServer().run(port);
50.     }
51. }
```

第15~16行首先添加`HttpServerCodec`，将请求和应答消息编码或者解码为HTTP消息；第17~18行增加`HttpObjectAggregator`，它的目的是将HTTP消息的多个部分组合成一条完整的HTTP消息；第19~20行添加`ChunkedWriteHandler`，来向客户端发送HTML5文件，它主要用于支持浏览器和服务端进行WebSocket通信；最后21~22行增加WebSocket服务端handler。

看了WebSocket的服务启动类，很多读者会心存疑惑：怎么WebSocket服务端的代码跟HTTP协议的非常类似呢？没有看到在`ChannelPipeline`中增加WebSocket的Handler，那如何处理WebSocket消息？这个疑问很好，下面就一起来从`WebSocketServerHandler`的实现中寻找答案。

代码清单11-2 WebSocket服务端处理类 `WebSocketServerHandler`

```
1.    public class WebSocketServerHandler extends SimpleChann
2.        private static final Logger logger = Logger
3.            .getLogger(WebSocketServerHandler.class.getName
4.
5.        private WebSocketServerHandshaker handshaker;
6.
7.        @Override
8.        public void messageReceived(ChannelHandlerContext c
9.            throws Exception {
10.        // 传统的HTTP接入
11.        if (msg instanceof FullHttpRequest) {
12.            handleHttpRequest(ctx, (FullHttpRequest) msg);
13.        }
14.        // WebSocket接入
15.        else if (msg instanceof WebSocketFrame) {
16.            handleWebSocketFrame(ctx, (WebSocketFrame) msg)
17.        }
18.        }
19.
20.        @Override
21.        public void channelReadComplete(ChannelHandlerConte
22.        ctx.flush();
23.        }
24.
25.        private void handleHttpRequest(ChannelHandlerContext
26.            FullHttpRequest req) throws Exception {
```

```

27.
28.     // 如果HTTP解码失败，返回HTTP异常
29.     if (!req.getDecoderResult().isSuccess()
30.         || (!"websocket".equals(req.headers().get("Upgr
31.         sendHttpResponse(ctx, req, new DefaultFullHttpR
32.             BAD_REQUEST));
33.         return;
34.     }
35.
36.     // 构造握手响应返回，本机测试
37.     WebSocketServerHandshakerFactory wsFactory = new We
38.         "ws://localhost:8080/websocket", null, false);
39.     handshaker = wsFactory.newHandshaker(req);
40.     if (handshaker == null) {
41.         WebSocketServerHandshakerFactory
42.             .sendUnsupportedWebSocketVersionResponse(ct
43.     } else {
44.         handshaker.handshake(ctx.channel(), req);
45.     }
46. }
47.
48. private void handleWebSocketFrame(ChannelHandlerCon
49.     WebSocketFrame frame) {
50.
51.     // 判断是否是关闭链路的指令
52.     if (frame instanceof CloseWebSocketFrame) {
53.         handshaker.close(ctx.channel(),

```



```
54.         (CloseWebSocketFrame) frame.retain());
55.     return;
56. }
57. // 判断是否是Ping消息
58. if (frame instanceof PingWebSocketFrame) {
59.     ctx.channel().write(
60.         new PongWebSocketFrame(frame.content().retain());
61.     return;
62. }
63. // 本例程仅支持文本消息，不支持二进制消息
64. if (!(frame instanceof TextWebSocketFrame)) {
65.     throw new UnsupportedOperationException(String.
66.         format("%s frame types not supported", frame.getClass().getName()));
67. }
68.
69. // 返回应答消息
70. String request = ((TextWebSocketFrame) frame).text();
71. if (logger.isLoggable(Level.FINE)) {
72.     logger.fine(String.format("%s received %s", ctx.channel(), request));
73. }
74. ctx.channel().write(
75.     new TextWebSocketFrame(request + " , 欢迎使用Netty WebSocket服务, 现在时刻: "
76.         + new java.util.Date().toString()));
77.
78. }
79.
80. private static void sendHttpResponse(ChannelHandler
```

```
81.         FullHttpRequest req, FullHttpResponse res) {
82.     // 返回应答给客户端
83.     if (res.getStatus().code() != 200) {
84.         ByteBuf buf = Unpooled.copiedBuffer(res.getStat
85.             CharsetUtil.UTF_8);
86.         res.content().writeBytes(buf);
87.         buf.release();
88.         setContentLength(res, res.content().readableByt
89.     }
90.
91.     // 如果是非Keep-Alive, 关闭连接
92.     ChannelFuture f = ctx.channel().writeAndFlush(res);
93.     if (!isKeepAlive(req) || res.getStatus().code() !=
94.         f.addListener(ChannelFutureListener.CLOSE);
95.     }
96. }
97.
98. @Override
99. public void exceptionCaught(ChannelHandlerContext c
100.     throws Exception {
101.     cause.printStackTrace();
102.     ctx.close();
103. }
104. }
```

首先从第11行看起，第一次握手请求消息由HTTP协议承载，所以

它是一个HTTP消息，执行handleHttpRequest方法来处理WebSocket握手请求。第29～34行首先对握手请求消息进行判断，如果消息头中没有包含Upgrade字段或者它的值不是websocket，则返回HTTP 400响应。

握手请求简单校验通过之后，开始构造握手工厂，创建握手处理类WebSocketServer Handshaker，通过它构造握手响应消息返回给客户端，同时将WebSocket相关的编码和解码类动态添加到ChannelPipeline中，用于WebSocket消息的编解码，代码如图11-10所示。

图11-10 WebSocket握手应答时动态增加编解码handler

添加WebSocket Encoder和WebSocket Decoder之后，服务端就可以自动对WebSocket消息进行编解码了，后面的业务handler可以直接对WebSocket对象进行操作。下面继续分析链路建立成功之后的操作：客户端通过文本框提交请求消息给服务端，WebSocket ServerHandler接收到的是已经解码后的WebSocketFrame消息。第48～96行对WebSocket请求消息进行处理，首先需要对控制帧进行判断，如果是关闭链路的控制消息，就调用WebSocketServerHandshaker的close方法关闭WebSocket连接；如果是维持链路的Ping消息，则构造Pong消息返回。由于本例程的WebSocket通信双方使用的都是文本消息，所以对请求消息的类型进行判断，不是文本的抛出异常。

最后，从TextWebSocketFrame中获取请求消息字符串，对它处理后通过构造新的TextWebSocketFrame消息返回给客户端，由于握手应答时动态增加了TextWebSocketFrame的编码类，所以，可以直接发送TextWebSocketFrame对象。

客户端浏览器接收到服务端的应答消息后，将其内容取出展示到浏

览器页面中。我们简单分析客户端WebSocketServer.html的源码。

```
<html>
<head>
<meta charset="UTF-8">
Netty WebSocket 时间服务器
</head>
<br>
<body>
<br>
<script type="text/javascript">
var

socket;

    if

(!window.WebSocket)
    {
        window.WebSocket = window.MozWebSocket;
    }
    if

(window.WebSocket) {
```

```
socket = new WebSocket("ws://localhost:8080/websocket");
```

```
socket.onmessage = function(event) {
```

```
    var ta = document.getElementById('responseText');
```

```
    ta.value="";
```

```
    ta.value = event.data
```

```
};
```

```
socket.onopen = function(event) {
```

```
var ta = document.getElementById('responseText');
```

```
ta.value = "打开WebSocket服务正常，浏览器支持WebSocket!";
```

```
};
```

```
socket.onclose = function(event) {
```

```
var ta = document.getElementById('responseText');
```

```
ta.value = "";
```

```
ta.value = "WebSocket 关闭!";
```

```
};
```

```
}
```

```
else
```

```
{
```

```
    alert("抱歉，您的浏览器不支持WebSocket协议!");
```

```
}
```

```
function
```

```
send(message) {
```

```
    if (!window.WebSocket) { return; }
```

```
    if (socket.readyState == WebSocket.OPEN) {
```

```
socket.send(message);
```

```
}
```

```
else
```

```
{
```

```
    alert("WebSocket连接没有建立成功!");
```

```
}
```

```
}
```

```
</script>
```

```
<form onsubmit="return false;">
```

```
<input type="text"
```

```
name="message"
```

```
value="Netty最佳实践"
```

```
/>
```

```
<br><br>
```

```
<input type="button"
```



```
value="发送WebSocket请求消息"
```

```
onclick="send (this.form.message.value)"/>
```

```
<hr color="blue"
```

```
/>
```

```
<h3>服务端返回的应答消息</h3>
```

```
<textarea id="responseText"
```

```
style="width:500px;
```

```
height:300px;
```

```
"></textarea>
```

```
</form>
```

```
</body>
```

```
</html>
```

由于通过JS开发WebSocket的接口比较简单，所以此处不再赘述，

对此感兴趣的同学可以通过JSR356、相关的技术书籍或者网站进行学习。

好，基于Netty的WebSocket服务端已经开发完毕，下个小节我们一起来测试下本节开发的程序，看它的各项功能是否能够达到设计预期。

11.3.3 运行WebSocket服务端

启动WebSocket服务，采用支持WebSocket的浏览器访问WebSocketServer.html，显示结果如图11-11所示。

图11-11 通过浏览器访问WebSocket服务端

文本框中显示“打开WebSocket服务正常，浏览器支持WebSocket!”，说明WebSocket链路建立成功，然后单击【发送WebSocket请求消息】按钮，显示结果如图11-12所示。

图11-12 打印WebSocket服务端返回的结果

当使用不支持WebSocket的老版本IE浏览器打开WebSocketServer.html时，运行结果如图11-13所示。

图11-13 不支持WebSocket的浏览器运行效果图

11.4 总结

本章首先介绍了HTTP协议的弊端和产生WebSocket的一些技术背景，随后对WebSocket的优势和基础入门知识进行了介绍，包括WebSocket的握手请求和响应、连接的建立和关闭、WebSocket的生命周期等。

学习了WebSocket的基础知识之后，通过Netty WebSocket时间服务器的开发，读者朋友可以更好地掌握如何利用Netty提供的WebSocket协议栈进行WebSocket应用程序的开发。

由于WebSocket本身的复杂性，以及可以通过多种形式（例如文本方式、二进制方式）承载消息，所以，它的API和用法也非常多，限于篇幅，本书无法对这些场景一一枚举。希望本章的例程能够起到抛砖引玉的作用，想要掌握更多的用法，需要读者结合Netty的测试用例和示例，以及WebSocket相关类库的Java Doc进行深入学习和实践，相信通过不断的实践很快就能掌握更多的功能和用法。

下一章，我们将继续学习如何利用Netty进行UDP协议的开发。

第12章 UDP协议开发

UDP是用户数据报协议（User Datagram Protocol，UDP）的简称，其主要作用是将网络数据流量压缩成数据报形式，提供面向事务的简单信息传送服务。与TCP协议不同，UDP协议直接利用IP协议进行UDP数据报的传输，UDP提供的是面向无连接的、不可靠的数据报投递服务。当使用UDP协议传输信息时，用户应用程序必须负责解决数据报丢失、重复、排序，差错确认等问题。

由于UDP具有资源消耗小、处理速度快的优点，所以通常视频、音频等可靠性要求不高的数据传输一般会使用UDP，即便有一定的丢包率，也不会对功能造成严重的影响。

从本章开始，我们将学习如何利用Netty开发异步的UDP应用程序。

本章主要包括：

- UDP协议简介
- UDP服务端开发
- UDP客户端开发
- 运行UDP例程

12.1 UDP协议简介

UDP是无连接的，通信双方不需要建立物理链路连接。在网络中它用于处理数据包，在OSI模型中，它处于第四层传输层，即位于IP协议的上一层。它不对数据报分组、组装、校验和排序，因此是不可靠的。报文的发送者不知道报文是否被对方正确接收。

UDP数据报格式有首部和数据两个部分，首部很简单，为8个字节，包括以下部分。

- (1) 源端口：源端口号，2个字节，最大值为65535；
- (2) 目的端口：目的端口号，2个字节，最大值为65535；
- (3) 长度：2字节，UDP用户数据报的总长度；

(4) 校验和：2字节，用于校验UDP数据报的数字段和包含UDP数据报首部的“伪首部”。其校验方法类似于IP分组首部中的首部校验和。

伪首部，又称为伪包头（Pseudo Header）：是指在TCP的分段或UDP的数据报格式中，在数据报首部前面增加源IP地址、目的IP地址、IP分组的协议字段、TCP或UDP数据报的总长度等，共12字节，所构成的扩展首部结构。此伪首部是一个临时的结构，它既不向上也不向下传递，仅仅是为了保证可以校验套接字的正确性。

UDP协议数据报格式示意图如图12-1所示。

图12-1 UDP协议数据报格式

UDP协议的特点如下。

(1) UDP传送数据前并不与对方建立连接，即UDP是无连接的。在传输数据前，发送方和接收方相互交换信息使双方同步；

(2) UDP对接收到的数据报不发送确认信号，发送端不知道数据是否被正确接收，也不会重发数据；

(3) UDP传送数据比TCP快速，系统开销也少：UDP比较简单，UDP头包含了源端口、目的端口、消息长度和校验和等很少的字节。由于UDP比TCP简单、灵活，常用于可靠性要求不高的数据传输，如视频、图片以及简单文件传输系统（TFTP）等。TCP则适用于可靠性要求很高但实时性要求不高的应用，如文件传输协议FTP、超文本传输协议HTTP、简单邮件传输协议SMTP等。

在下面的章节我们一起学习如何通过Netty开发UDP协议客户端和服务端应用。

12.2 UDP服务端开发

由于UDP通信双方不需要建立链路，所以，代码相对于TCP更加简单一些，下面来看服务端代码。

代码清单12-1 UDP服务端启动类 ChineseProverbServer

```
1.  public class ChineseProverbServer {
2.      public void run(int port) throws Exception {
3.          EventLoopGroup group = new NioEventLoopGroup();
4.          try {
5.              Bootstrap b = new Bootstrap();
6.              b.group(group).channel(NioDatagramChannel.class)

7.                  .option(ChannelOption.SO_BROADCAST, true)

8.                  .handler(new ChineseProverbServerHandler());

9.              b.bind(port).sync().channel().closeFuture().await()
10.         } finally {
```

```
11.         group.shutdownGracefully();
12.     }
13. }
14.
15.     public static void main(String[] args) throws Excep
16.     int port = 8080;
17.     if (args.length > 0) {
18.         try {
19.             port = Integer.parseInt(args[0]);
20.         } catch (NumberFormatException e) {
21.             e.printStackTrace();
22.         }
23.     }
24.     new ChineseProverbServer().run(port);
25. }
26. }
```

首先看第6行，由于使用UDP通信，在创建Channel的时候需要通过NioDatagramChannel来创建，随后设置Socket参数支持广播，最后设置业务处理handler。

相比于TCP通信，UDP不存在客户端和服务端的实际连接，因此不需要为连接（ChannelPipeline）设置handler，对于服务端，只需要设置启动辅助类的handler即可。

下面看ChineseProverbServerHandler的实现。

代码清单12-2 UDP服务端启动类 ChineseProverbServerHandler

```
1.    public class ChineseProverbServerHandler extends
2.        SimpleChannelInboundHandler<DatagramPacket> {
3.        // 谚语列表
4.        private static final String[] DICTIONARY={"只要功夫深
5.            "旧时王谢堂前燕，飞入寻常百姓家。", "洛阳亲友如相问，一
6.            "老骥伏枥，志在千里。烈士暮年，壮心不已!" };
7.
8.        private String nextQuote() {
9.            int quoteId=ThreadLocalRandom.current().nextInt(DIC
10.            return DICTIONARY[quoteId];
11.        }
12.
13.        @Override
14.        public void messageReceived(ChannelHandlerContext c
15.            throws Exception {
16.            String req = packet.content().toString(CharsetUtil.
17.            System.out.println(req);
18.            if ("谚语字典查询?".equals(req)) {
19.                ctx.writeAndFlush(new DatagramPacket(Unpooled.c
20.                    "谚语查询结果：" + nextQuote(), CharsetUtil.U
21.                    .sender()));
22.            }
23.        }
24.
```

```
25.         @Override
26.         public void exceptionCaught(ChannelHandlerContext c
27.             throws Exception {
28.             ctx.close();
29.             cause.printStackTrace();
30.         }
31.     }
```

首先看第14行，Netty对UDP进行了封装，因此，接收到的是Netty封装后的`io.netty.channel.socket.DatagramPacket`对象。第16行将`packet`内容转换为字符串（利用`ByteBuf`的`toString(Charset)`方法），然后对请求消息进行合法性判断：如果是“谚语字典查询？”，则构造应答消息返回。`DatagramPacket`有两个参数：第一个是需要发送的内容，为`ByteBuf`；另一个是目的地址，包括IP和端口，可以直接从发送的报文`DatagramPacket`中获取。

由于`ChineseProverbServerHandler`存在多线程并发操作的可能，所以使用了Netty的线程安全随机类`ThreadLocalRandom`。如果使用的是JDK7，可以直接使用JDK7的`java.util.concurrent.ThreadLocalRandom`。

下面我们来简单回顾下UDP服务端处理流程图，如图12-2所示。

图12-2 UDP服务端处理流程图


```
16.         if (!ch.closeFuture().await(15000)) {
17.             System.out.println("查询超时!");
18.         }
19.     } finally {
20.         group.shutdownGracefully();
21.     }
22. }
23.
24. public static void main(String[] args) throws Excep
25.     int port = 8080;
26.     if (args.length > 0) {
27.         try {
28.             port = Integer.parseInt(args[0]);
29.         } catch (NumberFormatException e) {
30.             e.printStackTrace();
31.         }
32.     }
33.     new ChineseProverbClient().run(port);
34. }
35. }
```

创建UDP Channel和设置支持广播属性等与服务端完全一致。由于不需要和服务端建立链路，UDP Channel创建完成之后，客户端就要主动发送广播消息；TCP客户端是在客户端和服务端链路建立成功之后由客户端的业务handler发送消息，这就是两者最大的区别。

第12和15行用于构造DatagramPacket发送广播消息，注意，广播消

息的IP设置为“255.255.255.255”。消息广播之后，客户端等待15s用于接收服务端的应答消息，然后退出并释放资源。

下面继续看客户端handler类的实现。

代码清单12-4 UDP客户端处理类 ChineseProverbClientHandler

```
1.    public class ChineseProverbClientHandler extends
2.        SimpleChannelInboundHandler<DatagramPacket> {
3.
4.        @Override
5.        public void messageReceived(ChannelHandlerContext c
6.            throws Exception {
7.            String response = msg.content().toString(CharsetUti
8.            if (response.startsWith("谚语查询结果: ")) {
9.                System.out.println(response);
10.               ctx.close();
11.            }
12.        }
13.
14.        @Override
15.        public void exceptionCaught(ChannelHandlerContext c
16.            throws Exception {
17.            cause.printStackTrace();
18.            ctx.close();
19.        }
20.    }
```

代码非常简单，接收到服务端的消息之后将其转成字符串，然后判断是否以“谚语查询结果：”开头，如果没有发生丢包等问题，数据是完整的，就打印查询结果，然后释放资源。

12.4 运行UDP例程

首先启动UDP服务端，然后启动客户端（运行两次），查看运行结果。

服务端运行结果如图12-3所示。

图12-3 UDP服务端运行结果

客户端运行结果1如图12-4所示。

图12-4 UDP客户端运行结果1

客户端运行结果2如图12-5所示。

图12-5 UDP客户端运行结果2

通过客户端的两次运行结果可以看出，每次运行的结果都不一样，说明UDP服务端的谚语查询功能正确。客户端能够成功接收并处理服务端的应答，说明在这个过程中没有发生丢包和乱序等问题。

12.5 总结

本章详细介绍了如何利用Netty进行UDP服务端和客户端的开发。首先对UDP协议进行了简单介绍，随后通过Netty UDP服务端和客户端的开发以及运行让读者尽快地掌握UDP开发的步骤，熟悉相关类库的应用。

由于UDP相对于TCP应用领域更窄一些，所以，本书不把UDP作为重点进行介绍，如果读者所从事的工作跟UDP强相关，或者希望对UDP有更加深入的了解，可以通过其他UDP相关的书籍进行深入学习。

第13章 文件传输

文件（File）是最常见的数据源之一，在程序中经常需要将数据存储到文件中，例如图片文件、声音文件等数据文件。在实际使用时，文件都包含一个特定的格式，这个格式需要程序员根据需求进行设计。读取已有的文件时也需要熟悉对应的文件格式，才能把数据从文件中正确地读取出来。

在NIO类库提供之前，Java所有的文件操作分为两大类：

- 基于字节流的InputStream和OutputStream；
- 基于字符流的Reader和Writer。

通过NIO新提供的FileChannel类库可以方便地以“管道”方式对文件进行各种I/O操作，相比于传统以流的方式进行的I/O操作有了很大的变化和进步。从本章开始，我们将学习如何通过Netty进行文件I/O操作。

本章主要包括：

- 文件的基础知识
- Netty文件传输开发
- 运行Netty文件传输服务例程

13.1 文件的基础知识

在开始学习Netty文件传输应用之前，我们首先对文件的基础知识进行介绍，如果读者对文件系统和Java文件操作已经非常熟悉，建议跳过本小节，直接学习Netty的文件传输应用开发。

13.1.1 文件的概念

文件是计算机中一种基本的数据存储形式，在实际存储数据时，如果对于数据的读写速度要求不是很高，存储的数据量不是很大，使用文件作为一种持久数据存储的方式是比较好的选择。存储在文件内部的数据和内存中的数据不同，存储在文件中的数据是一种“持久存储”，也就是当程序退出或计算机关机以后，数据还是存在的，而内存中的数据在程序退出或计算机关机以后，就会丢失了。

在不同的存储介质中，文件中的数据都是以一定的顺序依次存储起来的。在实际读取时由硬件以及操作系统完成对于数据的控制，保证程序读取到的数据和存储的顺序一致。每个文件以一个文件路径和文件名称进行表示，在需要访问该文件时，只需要知道该文件的路径以及文件的全名即可。在不同的操作系统环境下，文件路径的表示形式是不一样的，例如在Windows操作系统中一般的表示形式为C:\windows，而在Linux上的表示形式为/home/lilinfeng。

13.1.2 文件路径

绝对路径是指文件的完整路径，例如
D:\java\nio\netty\FileServer.java，该路径中包含文件的完整路径
D:\java\nio\netty以及文件的全名FileServer.java。使用该路径可以找到一

个唯一的文件。使用绝对路径的最大缺点就是不同的操作系统文件路径和表示形式不同，使用不当往往会导致文件读取失败，因此，在实际项目应用时，往往使用相对路径或者类路径。

在Eclipse项目中运行程序时，当前路径是项目的根目录。假如，工作空间存储在D:\java\nio，当前项目名称是netty，则当前路径是D:\java\nio\netty。在Eclipse控制台运行程序时，当前路径是class文件所在的目录，如果class文件包含包名，则以该class文件最顶层的包名作为当前路径。

13.1.3 文件名称

文件一般采用“文件名.后缀名”的形式进行命名，其中“文件名”用来表示文件的作用，而使用后缀名来表示文件的类型，这是当前操作系统中常见的一种形式。例如“API.doc”文件，其中API代表该文件的名称，而后缀名doc代表文件是windows office word类型，在Windows操作系统中，还会自动将特定格式的后缀名和对应的程序关联，在双击该文件时使用特定的程序打开。

文件名称只是一个标示，和实际存储的文件内容没有必然的联系。在程序中需要存储数据时，如果自己设计了特定的文件格式，则可以自定义文件的后缀名，来表示文件类型。和文件路径一样，在Java代码内部书写文件名称时也区分大小写，文件名称的大小写必须和操作系统中的大小写保持一致。

13.1.4 FileChannel简介

Java NIO中的FileChannel是一个连接到文件的通道,可以通过这个文

件通道读写文件。JDK1.7之前NIO1.0的FileChannel是同步阻塞的，JDK1.7版本对NIO类库进行了升级，升级后的NIO2.0提供了异步文件通道AsynchronousFileChannel，它支持异步非阻塞文件操作（AIO）。

在使用FileChannel之前必须先打开它，FileChannel无法直接被打开，需要通过使用InputStream、OutputStream或RandomAccessFile来获取一个FileChannel实例。下面示范如何通过RandomAccessFile打开FileChannel。

```
RandomAccessFile billFile = new RandomAccessFile("home/lilinf  
FileChannel channel = billFile.getChannel();
```

如果需要从FileChannel中读取数据，要申请一个ByteBuffer，将数据从FileChannel中读取到字节缓冲区中。read()方法返回的int值表示有多少字节被读到了字节缓冲区中，如果返回-1，表示读到了文件末尾。

如果需要通过FileChannel向文件中写入数据，需要将数据复制或者直接存放到ByteBuffer中，然后调用FileChannel.write()方法进行写操作。示例代码如下。

```
String content = "13888888888|北京市海淀区|全球通|VIP用户|";  
ByteBuffer writeBuffer = ByteBuffer.allocate(128);  
writeBuffer.put(content.getBytes());  
writeBuffer.flip();  
channel.write(buf);
```

使用完FileChannel之后，需要通过close()方法关闭文件句柄，防止出现句柄泄漏。

可以通过FileChannel的position(long pos)方法设置文件的位置指针，利用该特性可以实现文件的随机读写。

13.2 Netty文件传输开发

在实际项目中，文件传输通常采用FTP或者HTTP附件的方式。事实上通过TCP Socket+File的方式进行文件传输也有一定的应用场景，尽管不是主流，但是掌握这种文件传输方式还是比较重要的，特别是针对两个跨主机的JVM进程之间进行持久化数据的相互交换。

下面我们一起来看看如何通过Netty的NIO类库进行文件传输。在10.3.3章节我们介绍JiBx的时候，曾经开发了一个Ant脚本用于生成POJO对象和XML的绑定关系文件binding.xml，文件传输例程就选择此文件作为待传输的文件。

具体场景如下。

- (1) Netty文件服务器启动，绑定8080作为内部监听端口；
- (2) 在CMD控制台，通过telnet和文件服务器建立TCP连接；
- (3) 控制台输入需要下载的文件绝对路径；
- (4) 文件服务器接收到请求消息后进行合法性判断，如果文件存在，则将文件发送给CMD控制台；
- (5) CMD控制台打印文件名和文件内容。

首先看下服务端启动类的实现。

代码清单13-1 文件服务端启动类 FileServer

[illegible]

```
23.                new StringDecoder(CharsetUtil.UTF_8),

24.                new FileServerHandler());

25.        }
26.    });
27.    ChannelFuture f = b.bind(port).sync();
28.    System.out.println("Start file server at port : " + port);
29.    f.channel().closeFuture().sync();
30. } finally {
31.     // 优雅停机
32.     bossGroup.shutdownGracefully();
33.     workerGroup.shutdownGracefully();
34. }
35. }
36.
37.     public static void main(String[] args) throws Exception {
38. int port = 8080;
39. if (args.length > 0) {
40.     try {
41.         port = Integer.parseInt(args[0]);
42.     } catch (NumberFormatException e) {
```

```
1. public class FileServerHandler extends SimpleChannelInboundHandler<File>() {
2.
3.     private static final String CR = System.getProperty("\n");
4.
5.     /**
6.      * (non-Javadoc)
7.      *
8.      * @see
9.      * io.netty.channel.SimpleChannelInboundHandler#messageReceived(ChannelHandlerContext,

```

```

10.      * .channel.ChannelHandlerContext, java.lang.Object)
11.      */
12.      public void messageReceived(ChannelHandlerContext ctx,
13.      throws Exception {
14.  File file = new File(msg);
15.  if (file.exists()) {
16.      if (!file.isFile()) {
17.          ctx.writeAndFlush("Not a file : " + file + CR);
18.          return;
19.      }
20.      ctx.write(file + " " + file.length() + CR);
21.      RandomAccessFile randomAccessFile = new RandomAccessFile(
22.      FileRegion region = new DefaultFileRegion(
23.          randomAccessFile.getChannel(), 0, randomAccessFile.length());
24.      ctx.write(region);
25.      ctx.writeAndFlush(CR);
26.      randomAccessFile.close();
27.  } else {
28.      ctx.writeAndFlush("File not found: " + file + CR);
29.  }
30.  }
31.
32.  /*
33.   * (non-Javadoc)
34.   *
35.   * @see
36.   * io.netty.channel.ChannelHandlerAdapter#exceptionCaught()

```

```
37.         * .ChannelHandlerContext, java.lang.Throwable)
38.         */
39.     public void exceptionCaught(ChannelHandlerContext ctx,
40.         throws Exception {
41. cause.printStackTrace();
42. ctx.close();
43.     }
44. }
```

第15~19行首先对文件的合法性进行校验，如果不存在，构造异常消息返回。如果文件存在，使用RandomAccessFile以只读的方式打开文件，然后通过Netty提供的DefaultFileRegion进行文件传输，它有如下三个参数。

- FileChannel：文件通道，用于对文件进行读写操作；
- Position：文件操作的指针位置，读取或者写入的起始点；
- Count：操作的总字节数。

构造完DefaultFileRegion之后，可以直接调用ChannelHandlerContext的write方法实现文件的发送。Netty底层对文件写入进行了封装，上层应用不需要关心发送的细节。最后写入回车换行符告知CMD控制台：文件传输结束。

13.3 运行Netty文件传输服务例程

启动文件服务器，打开CMD控制台，通过telnet命令连接到主机，如图13-1所示。

图13-1 通过telnet连接文件服务器

连接成功之后，输入需要传输的文件路径：“M:\software\eclipse-SDK-4.2.2-win32\ eclipse\workspace\book\binding.xml”，然后输入回车键，显示结果如图13-2所示。

图13-2 文件服务器返回结果

下面我们继续输入一个不存在的文件路径，运行结果如图13-3所示。

图13-3 文件服务器校验失败场景

通过以上两个用例的运行结果可以看出，Netty的文件服务器功能运行正常，可以实现文件的正确传输。

13.4 总结

本章介绍了如何利用Netty进行文件传输。由于Netty对文件传输进行了封装，上层业务应用不需要感知文件操作的细节，由于Netty提供了多种编解码类库，通过组合可以灵活地处理各种文件。

事实上，Netty有多种方式可以实现文件的传输，例程仅仅给出了一种最为通用的实现方式。在进行大文件传输的时候，一次将文件内容全部映射到内存中，很有可能导致内存溢出。为了解决大文件传输过程中的内存溢出，Netty提供了ChunkedWriteHandler来解决大文件或者码流传输过程中可能发生的内存溢出问题。Netty的文件传输无论在功能还是可靠性方面，相比于传统的I/O类库或者其他一些第三方文件传输类库，都有独特的优势。

由于文件传输不是本书的重点，如果读者朋友想要了解更多的Netty传输的相关知识，可以通过阅读ChunkedWriteHandler、FileRegion等类库的API DOC和源码，在学习和实践中掌握更多的高级用法和功能。

第14章 私有协议栈开发

通信协议从广义上区分，可以分为公有协议和私有协议。由于私有协议的灵活性，它往往会在某个公司或者组织内部使用，按需定制，也因为如此，升级起来会非常方便，灵活性好。

绝大多数的私有协议传输层都基于TCP/IP，所以利用Netty的NIO TCP协议栈可以非常方便地进行私有协议的定制和开发。本章节通过一个私有协议的设计和开发，让读者能够熟悉和掌握这方面的知识。

本章主要包括：

- 私有协议介绍
- 基于Netty的私有协议栈设计
- 私有协议栈开发

14.1 私有协议介绍

私有协议本质上是厂商内部发展和采用的标准，除非授权，其他厂商一般无权使用该协议。私有协议也称非标准协议，就是未经国际或国家标准化组织采纳或批准，由某个企业自己制订，协议实现细节不愿公开，只在企业自己生产的设备之间使用的协议。私有协议具有封闭性、垄断性、排他性等特点。如果网上大量存在私有（非标准）协议，现行网络或用户一旦使用了它，后进入的厂家设备就必须跟着使用这种非标准协议，才能够互连互通，否则根本不可能进入现行网络。这样，使用非标准协议的厂家就实现了垄断市场的愿望。

尽管私有协议具有垄断性的特征，但并非所有的私有协议设计者的初衷就是为了垄断。由于现代软件系统的复杂性，一个大型软件系统往往会被人为地拆分成多个模块，另外随着移动互联网的兴起，网站的规模也越来越大，业务的功能越来越多，为了能够支撑业务的发展，往往需要集群和分布式部署，这样，各个模块之间就要进行跨节点通信。

在传统的Java应用中，通常使用以下4种方式进行跨节点通信。

- （1）通过RMI进行远程服务调用；
- （2）通过Java的Socket+Java序列化的方式进行跨节点调用；
- （3）利用一些开源的RPC框架进行远程服务调用，例如Facebook的Thrift，Apache的Avro等；
- （4）利用标准的公有协议进行跨节点服务调用，例如HTTP+XML、RESTful+JSON或者WebService。

跨节点的远程服务调用，除了链路层的物理连接外，还需要对请求和响应消息进行编解码。在请求和应答消息本身以外，也需要携带一些其他控制和管理类指令，例如链路建立的握手请求和响应消息、链路检测的心跳消息等。当这些功能组合到一起之后，就会形成私有协议。

事实上，私有协议并没有标准的定义，只要是能够用于跨进程、跨主机数据交换的非标准协议，都可以称为私有协议。通常情况下，正规的私有协议都有具体的协议规范文档，类似于《XXXX协议VXX规范》，但是在实际的项目中，内部使用的私有协议往往是口头约定的规范，由于并不需要对外呈现或者被外部调用，所以一般不会单独写相关的内部私有协议规范文档。

本章使用Netty提供的异步TCP协议栈开发一个私有协议栈，该协议栈被命名为Netty协议栈。从下个小节开始，我们将详细介绍Netty协议的设计和开发。

14.2 Netty协议栈功能设计

Netty协议栈用于内部各模块之间的通信，它基于TCP/IP协议栈，是一个类HTTP协议的应用层协议栈，相比于传统的标准协议栈，它更加轻巧、灵活和实用。

14.2.1 网络拓扑图

如图14-1所示，在分布式组网环境下，每个Netty节点（Netty进程）之间建立长连接，使用Netty协议进行通信。Netty节点并没有服务端和客户端的区分，谁首先发起连接，谁就作为客户端，另一方自然就成为服务端。一个Netty节点既可以作为客户端连接另外的Netty节点，也可以作为Netty服务端被其他Netty节点连接，这完全取决于使用者的业务场景和具体的业务组网。

图14-1 Netty协议网络拓扑示意图

14.2.2 协议栈功能描述

Netty协议栈承载了业务内部各模块之间的消息交互和服务调用，它的主要功能如下。

- （1）基于Netty的NIO通信框架，提供高性能的异步通信能力；
- （2）提供消息的编解码框架，可以实现POJO的序列化和反序列化；
- （3）提供基于IP地址的白名单接入认证机制；

(4) 链路的有效性校验机制；

(5) 链路的断连重连机制。

14.2.3 通信模型

Netty协议栈通信模型如图14-2所示。

图14-2 Netty协议栈通信交互图

(1) Netty协议栈客户端发送握手请求消息，携带节点ID等有效身份认证信息；

(2) Netty协议栈服务端对握手请求消息进行合法性校验，包括节点ID有效性校验、节点重复登录校验和IP地址合法性校验，校验通过后，返回登录成功的握手应答消息；

(3) 链路建立成功之后，客户端发送业务消息；

(4) 链路成功之后，服务端发送心跳消息；

(5) 链路建立成功之后，客户端发送心跳消息；

(6) 链路建立成功之后，服务端发送业务消息；

(7) 服务端退出时，服务端关闭连接，客户端感知对方关闭连接后，被动关闭客户端连接。

备注：需要指出的是，Netty协议通信双方链路建立成功之后，双方可以进行全双工通信，无论客户端还是服务端，都可以主动发送请求消息给对方，通信方式可以是TWO WAY或者ONE WAY。双方之间的

心跳采用Ping-Pong机制，当链路处于空闲状态时，客户端主动发送Ping消息给服务端，服务端接收到Ping消息后发送应答消息Pong给客户端，如果客户端连续发送N条Ping消息都没有接收到服务端返回的Pong消息，说明链路已经挂死或者对方处于异常状态，客户端主动关闭连接，间隔周期T后发起重连操作，直到重连成功。

14.2.4 消息定义

Netty协议栈消息定义包含两部分：

- 消息头；
- 消息体。

表14-1 Netty消息定义表（NettyMessage）

表14-2 Netty协议消息头定义（Header）

14.2.5 Netty协议支持的字段类型

表14-3 Netty协议支持的数据类型

14.2.6 Netty协议的编解码规范

1. Netty协议的编码

Netty协议NettyMessage的编码规范如下。

（1）crcCode: `java.nio.ByteBuffer.putInt(int value)`，如果采用其他缓冲区实现，必须与其等价；

(2) length: `java.nio.ByteBuffer.putInt(int value)`, 如果采用其他缓冲区实现, 必须与其等价;

(3) sessionID: `java.nio.ByteBuffer.putLong(long value)`, 如果采用其他缓冲区实现, 必须与其等价;

(4) type: `java.nio.ByteBuffer.put(byte b)`, 如果采用其他缓冲区实现, 必须与其等价;

(5) priority: `java.nio.ByteBuffer.put(byte b)`, 如果采用其他缓冲区实现, 必须与其等价;

(6) attachment: 它的编码规则为——如果attachment长度为0, 表示没有可选附件, 则将长度编码设为0, `java.nio.ByteBuffer.putInt(0)`; 如果大于0, 说明有附件需要编码, 具体的编码规则如下——

- 首先对附件的个数进行编码,
`java.nio.ByteBuffer.putInt(attachment.size());`
- 然后对Key进行编码, 先编码长度, 再将它转换成byte数组之后编码内容, 具体代码如下。

```
String key = null;
```

```
byte
```

```
[] value = null;
```

```
for
```

```
(Map.Entry<String, Object
```

```
> param : attachment.entrySet()) {  
    key = param.getKey();  
    buffer.writeString(key);  
    value = marshaller.writeObject(param.getValue());  
    buffer.writeBinary(value);  
}  
key = null;  
  
value = null;
```

需要说明的是，将String字符串写入ByteBuffer和通过Jboss

Marshalling将Object序列化为byte数组，此处没有详细展开介绍，后续代码开发章节会给出具体实现。

（7）body的编码：通过JBoss Marshalling将其序列化为byte数组，然后调用`java.nio.ByteBuffer.put(byte [] src)`将其写入ByteBuffer缓冲区中。

由于整个消息的长度必须等全部字段都编码完成之后才能确认，所以最后需要更新消息头中的length字段，将其重新写入ByteBuffer中。

2. Netty协议的解码

相对于NettyMessage的编码，仍旧以`java.nio.ByteBuffer`为例，给出Netty协议的解码规范。

（1）crcCode：通过`java.nio.ByteBuffer.getInt()`获取校验码字段，其他缓冲区需要与其等价；

（2）length：通过`java.nio.ByteBuffer.getInt()`获取Netty消息的长度，其他缓冲区需要与其等价；

（3）sessionId：通过`java.nio.ByteBuffer.getLong()`获取会话ID，其他缓冲区需要与其等价；

（4）type：通过`java.nio.ByteBuffer.get()`获取消息类型，其他缓冲区需要与其等价；

（5）priority：通过`java.nio.ByteBuffer.get()`获取消息优先级，其他缓冲区需要与其等价；

(6) attachment: 它的解码规则为——首先创建一个新的 attachment对象, 调用java.nio.ByteBuffer.getInt()获取附件的长度, 如果为0, 说明附件为空, 解码结束, 继续解消息体; 如果非空, 则根据长度通过for循环进行解码。

```
String key = null;
```

```
Object
```

```
value = null;
```

```
for
```

```
(int
```

```
i = 0; i < size; i++) {  
    key = buffer.readString();  
    value = unmarshaller.readObject(buffer.readBinary());  
    this.
```

```
attachment.put(key, value);  
    }  
    key = null;  
  
    value = null;
```

后续的代码开发章节会给出附件解码的具体实现，此处不再详细展开，仅仅给出解码的规则。

(7) body: 通过JBoss的marshaller对其进行解码。

14.2.7 链路的建立

Netty协议栈支持服务端和客户端，对于使用Netty协议栈的应用程序而言，不需要刻意区分到底是客户端还是服务端，在分布式组网环境中，一个节点可能既是服务端也是客户端，这个依据具体的用户场景而定。

Netty协议栈对客户端的说明如下：如果A节点需要调用B节点的服务，但是A和B之间还没有建立物理链路，则由调用方主动发起连接，此时，调用方为客户端，被调用方为服务端。

考虑到安全，链路建立需要通过基于IP地址或者号段的黑白名单安全认证机制，作为样例，本协议使用基于IP地址的安全认证，如果有多个IP，通过逗号进行分割。在实际商用项目中，安全认证机制会更加严格，例如通过密钥对用户名和密码进行安全认证。

客户端与服务端链路建立成功之后，由客户端发送握手请求消息，握手请求消息的定义如下。

- (1) 消息头的type字段值为3;
- (2) 可选附件个数为0;
- (3) 消息体为空;
- (4) 握手消息的长度为22个字节。

服务端接收到客户端的握手请求消息之后，如果IP校验通过，返回握手成功应答消息给客户端，应用层链路建立成功。握手应答消息定义如下。

- (1) 消息头的type字段值为4;
- (2) 可选附件个数为0;
- (3) 消息体为byte类型的结果，0：认证成功；-1：认证失败。

链路建立成功之后，客户端和服务端就可以互相发送业务消息了。

14.2.8 链路的关闭

由于采用长连接通信，在正常的业务运行期间，双方通过心跳和业

务消息维持链路，任何一方都不需要主动关闭连接。

但是，在以下情况下，客户端和服务端需要关闭连接。

（1）当对方宕机或者重启时，会主动关闭链路，另一方读取到操作系统的通知信号，得知对方REST链路，需要关闭连接，释放自身的句柄等资源。由于采用TCP全双工通信，通信双方都需要关闭连接，释放资源；

（2）消息读写过程中，发生了I/O异常，需要主动关闭连接；

（3）心跳消息读写过程中发生了I/O异常，需要主动关闭连接；

（4）心跳超时，需要主动关闭连接；

（5）发生编码异常等不可恢复错误时，需要主动关闭连接。

14.2.9 可靠性设计

Netty协议栈可能会运行在非常恶劣的网络环境中，网络超时、闪断、对方进程僵死或者处理缓慢等情况都有可能发生。为了保证在这些极端异常场景下Netty协议栈仍能够正常工作或者自动恢复，需要对它的可靠性进行统一规划和设计。

1. 心跳机制

在凌晨等业务低谷期时段，如果发生网络闪断、连接被Hang住等网络问题时，由于没有业务消息，应用进程很难发现。到了白天业务高峰期时，会发生大量的网络通信失败，严重的会导致一段时间进程内无法处理业务消息。为了解决这个问题，在网络空闲时采用心跳机制来检测

链路的互通性，一旦发现网络故障，立即关闭链路，主动重连。

具体的设计思路如下。

(1) 当网络处于空闲状态持续时间达到 T （连续周期 T 没有读写消息）时，客户端主动发送Ping心跳消息给服务端；

(2) 如果在下一个周期 T 到来时客户端没有收到对方发送的Pong心跳应答消息或者读取到服务端发送的其他业务消息，则心跳失败计数器加1；

(3) 每当客户端接收到服务的业务消息或者Pong应答消息，将心跳失败计数器清零；当连续 N 次没有接收到服务端的Pong消息或者业务消息，则关闭链路，间隔INTERVAL时间后发起重连操作；

(4) 服务端网络空闲状态持续时间达到 T 后，服务端将心跳失败计数器加1；只要接收到客户端发送的Ping消息或者其他业务消息，计数器清零；

(5) 服务端连续 N 次没有接收到客户端的Ping消息或者其他业务消息，则关闭链路，释放资源，等待客户端重连。

通过Ping-Pong双向心跳机制，可以保证无论通信哪一方出现网络故障，都能被及时地检测出来。为了防止由于对方短时间内繁忙没有及时返回应答造成的误判，只有连续 N 次心跳检测都失败才认定链路已经损害，需要关闭链路并重建链路。

当读或者写心跳消息发生I/O异常的时候，说明链路已经中断，此时需要立即关闭链路，如果是客户端，需要重新发起连接。如果是服务端，需要清空缓存的半包信息，等待客户端重连。

2. 重连机制

如果链路中断，等待INTERVAL时间后，由客户端发起重连操作，如果重连失败，间隔周期INTERVAL后再次发起重连，直到重连成功。

为了保证服务端能够有充足的时间释放句柄资源，在首次断连时客户端需要等待INTERVAL时间之后再发起重连，而不是失败后就立即重连。

为了保证句柄资源能够及时释放，无论什么场景下的重连失败，客户端都必须保证自身的资源被及时释放，包括但不限于SocketChannel、Socket等。

重连失败后，需要打印异常堆栈信息，方便后续的问题定位。

3. 重复登录保护

当客户端握手成功之后，在链路处于正常状态下，不允许客户端重复登录，以防止客户端在异常状态下反复重连导致句柄资源被耗尽。

服务端接收到客户端的握手请求消息之后，首先对IP地址进行合法性检验，如果校验成功，在缓存的地址表中查看客户端是否已经登录，如果已经登录，则拒绝重复登录，返回错误码-1，同时关闭TCP链路，并在服务端的日志中打印握手失败的原因。

客户端接收到握手失败的应答消息之后，关闭客户端的TCP连接，等待INTERVAL时间之后，再次发起TCP连接，直到认证成功。

为了防止由服务端和客户端对链路状态理解不一致导致的客户端无法握手成功的问题，当服务端连续N次心跳超时之后需要主动关闭链路，清空该客户端的地址缓存信息，以保证后续该客户端可以重连成功，防止被重复登录保护机制拒绝掉。

4. 消息缓存重发

无论客户端还是服务端，当发生链路中断之后，在链路恢复之前，缓存在消息队列中待发送的消息不能丢失，等链路恢复之后，重新发送这些消息，保证链路中断期间消息不丢失。

考虑到内存溢出的风险，建议消息缓存队列设置上限，当达到上限之后，应该拒绝继续向该队列添加新的消息。

14.2.10 安全性设计

为了保证整个集群环境的安全，内部长连接采用基于IP地址的安全认证机制，服务端对握手请求消息的IP地址进行合法性校验：如果在白名单之内，则校验通过；否则，拒绝对方连接。

如果将Netty协议栈放到公网中使用，需要采用更加严格的安全认证机制，例如基于密钥和AES加密的用户名+密码认证机制，也可以采用SSL/TSL安全传输。

作为示例程序，Netty协议栈采用最简单的基于IP地址的白名单安全认证机制。

14.2.11 可扩展性设计

Netty协议需要具备一定的扩展能力，业务可以在消息头中自定义业务域字段，例如消息流水号、业务自定义消息头等。通过Netty消息头中的可选附件attachment字段，业务可以方便地进行自定义扩展。

Netty协议栈架构需要具备一定的扩展能力，例如统一的消息拦截、接口日志、安全、加解密等可以被方便地添加和删除，不需要修改之前的逻辑代码，类似Servlet的Filter Chain和AOP，但考虑到性能因素，不推荐通过AOP来实现功能的扩展。

14.3 Netty协议栈开发

14.3.1 数据结构定义

首先，对Netty协议栈使用到的数据结构进行定义，Netty消息定义如下。

代码清单14-1 NettyMessage类定义

```
1. public final class NettyMessage {
2.     private Header header; //消息头
3.     private Object body; //消息体
4.
5.     /**
6.      * @return the header
7.      */
8.     public final Header getHeader() {
9. return header;
10.    }
11.
12.    /**
13.     * @param header
14.     *         the header to set
15.     */
16.    public final void setHeader(Header header) {
17. this.header = header;
```

```
18.     }
19.
20.     /**
21.      * @return the body
22.      */
23.     public final Object getBody() {
24. return body;
25.     }
26.
27.     /**
28.      * @param body
29.      *          the body to set
30.      */
31.     public final void setBody(Object body) {
32. this.body = body;
33.     }
34.
35.     /*
36.      * (non-Javadoc)
37.      *
38.      * @see java.lang.Object#toString()
39.      */
40.     @Override
41.     public String toString() {
42. return "NettyMessage [header=" + header + "];"
43.     }
44. }
```

代码清单14-2 消息头Header类定义

```
1. public final class Header {
2.     private int crcCode = 0xabef0101;
3.     private int length;// 消息长度
4.     private long sessionID;// 会话ID
5.     private byte type;// 消息类型
6.     private byte priority;// 消息优先级
7.     private Map<String, Object> attachment = new HashMap<String, Object>();
8.
9.     /**
10.      * @return the crcCode
11.      */
12.     public final int getCrcCode() {
13. return crcCode;
14.     }
15.
16.     /**
17.      * @param crcCode
18.      *         the crcCode to set
19.      */
20.     public final void setCrcCode(int crcCode) {
21. this.crcCode = crcCode;
22.     }
23. }
```

```
24.    /**
25.     * @return the length
26.     */
27.    public final int getLength() {
28. return length;
29.    }
30.
31.    /**
32.     * @param length
33.     *         the length to set
34.     */
35.    public final void setLength(int length) {
36. this.length = length;
37.    }
38.
39.    /**
40.     * @return the sessionID
41.     */
42.    public final long getSessionID() {
43. return sessionID;
44.    }
45.
46.    /**
47.     * @param sessionID
48.     *         the sessionID to set
49.     */
50.    public final void setSessionID(long sessionID) {
```

```
51. this.sessionID = sessionID;
52.     }
53.
54.     /**
55.      * @return the type
56.      */
57.     public final byte getType() {
58. return type;
59.     }
60.
61.     /**
62.      * @param type
63.      *          the type to set
64.      */
65.     public final void setType(byte type) {
66. this.type = type;
67.     }
68.
69.     /**
70.      * @return the priority
71.      */
72.     public final byte getPriority() {
73. return priority;
74.     }
75.
76.     /**
77.      * @param priority
```

```

78.      *           the priority to set
79.      */
80.      public final void setPriority(byte priority) {
81.  this.priority = priority;
82.  }
83.
84.      /**
85.      * @return the attachment
86.      */
87.      public final Map<String, Object> getAttachment() {
88.  return attachment;
89.  }
90.
91.      /**
92.      * @param attachment
93.      *           the attachment to set
94.      */
95.      public final void setAttachment(Map<String, Object> at
96.  this.attachment = attachment;
97.  }
98.
99.      /*
100.      * (non-Javadoc)
101.      *
102.      * @see java.lang.Object#toString()
103.      */
104.      @Override

```

```
105.    public String toString() {
106.        return "Header [crcCode=" + crcCode + ", length=" + length + ",
107.            + ", sessionID=" + sessionID + ", type=" + type +
108.            + priority + ", attachment=" + attachment + "];"
109.    }
110. }
```

由于心跳消息、握手请求和握手应答消息都可以统一由NettyMessage承载，所以不需要为这几类控制消息做单独的数据结构定义。

14.3.2 消息编解码

分别定义NettyMessageDecoder和NettyMessageEncoder用于NettyMessage消息的编解码，它们的具体实现如下。

代码清单14-3 Netty消息编码类：NettyMessageEncoder

```
1. public final class NettyMessageEncoder extends
2.    MessageToMessageEncoder<NettyMessage> {
3.
4.        MarshallingEncoder marshallEncoder;
5.
6.        public NettyMessageEncoder() throws IOException {
7.            this.marshallEncoder = new MarshallingEncoder();
8.        }
9.    }
```

```
10.     @Override
11.     protected void encode(ChannelHandlerContext ctx, NettyM
12.     List<Object> out) throws Exception {
13.
14.     if (msg == null || msg.getHeader() == null)
15.         throw new Exception("The encode message is null");
16.     ByteBuf sendBuf = Unpooled.buffer();
17.     sendBuf.writeInt((msg.getHeader().getCrcCode()));
18.     sendBuf.writeInt((msg.getHeader().getLength()));
19.     sendBuf.writeLong((msg.getHeader().getSessionID()));
20.     sendBuf.writeByte((msg.getHeader().getType()));
21.     sendBuf.writeByte((msg.getHeader().getPriority()));
22.     sendBuf.writeInt((msg.getHeader().getAttachment().size()));
23.     String key = null;
24.     byte[] keyArray = null;
25.     Object value = null;
26.     for (Map.Entry<String, Object> param : msg.getHeader().get
27.         .entrySet()) {
28.         key = param.getKey();
29.         keyArray = key.getBytes("UTF-8");
30.         sendBuf.writeInt(keyArray.length);
31.         sendBuf.writeBytes(keyArray);
32.         value = param.getValue();
33.         marshallEncoder.encode(value, sendBuf);
34.     }
35.     key = null;
36.     keyArray = null;
```

```
37. value = null;
38. if (msg.getBody() != null) {
39.     marshallEncoder.encode(msg.getBody(), sendBuf);
40. } else
41.     sendBuf.writeInt(0);
42.     sendBuf.setInt(4, sendBuf.readableBytes());
43. }
44. }
```

代码清单14-4 Netty消息编码工具类：MarshallEncoder

```
1. public class MarshallEncoder {
2.     private static final byte[] LENGTH_PLACEHOLDER = new byte[4];
3.     private Marshaller marshaller;
4.
5.     public MarshallEncoder() throws IOException {
6.         marshaller = MarshallingCodecFactory.buildMarshalling();
7.     }
8.
9.     protected void encode(Object msg, ByteBuf out) throws IOException {
10.    try {
11.        int lengthPos = out.writerIndex();
12.        out.writeBytes(LENGTH_PLACEHOLDER);
13.        ChannelBufferByteOutput output = new ChannelBufferByteOutput(out);
14.        marshaller.start(output);
15.        marshaller.writeObject(msg);
```

```
16.     marshaller.finish();
17.     out.setInt(lengthPos, out.writerIndex() - lengthPos -
18. } finally {
19.     marshaller.close();
20. }
21.     }
22. }
```

代码清单14-5 Netty消息解码类：NettyMessageDecoder

```
1. public class NettyMessageDecoder extends LengthFieldBasedFi
2.
3.     MarshallingDecoder marshallingDecoder;
4.
5.     public NettyMessageDecoder(int maxFrameLength, int lengt
6.         int lengthFieldLength) throws IOException {
7.         super(maxFrameLength, lengthFieldOffset, lengthFieldLengt
8.         marshallingDecoder = new MarshallingDecoder();
9.     }
10.
11.     @Override
12.     protected Object decode(ChannelHandlerContext ctx, Byte
13.         throws Exception {
14.         ByteBuf frame = (ByteBuf) super.decode(ctx, in);
15.         if (frame == null) {
16.             return null;
```



```
17. }
18.
19. NettyMessage message = new NettyMessage();
20. Header header = new Header();
21. header.setCrcCode(in.readInt());
22. header.setLength(in.readInt());
23. header.setSessionID(in.readLong());
24. header.setType(in.readByte());
25. header.setPriority(in.readByte());
26.
27. int size = in.readInt();
28. if (size > 0) {
29.     Map<String, Object> attch = new HashMap<String, Object>();
30.     int keySize = 0;
31.     byte[] keyArray = null;
32.     String key = null;
33.     for (int i = 0; i < size; i++) {
34.         keySize = in.readInt();
35.         keyArray = new byte[keySize];
36.         in.readBytes(keyArray);
37.         key = new String(keyArray, "UTF-8");
38.         attch.put(key, marshallDecoder.decode(in));
39.     }
40.     keyArray = null;
41.     key = null;
42.     header.setAttachment(attch);
43. }
```

```
44. if (in.readableBytes() > 4) {
45.     message.setBody(marshallingDecoder.decode(in));
46. }
47. message.setHeader(header);
48. return message;
49. }
50. }
```

在这里我们用到了Netty的LengthFieldBasedFrameDecoder解码器，它支持自动的TCP粘包和半包处理，只需要给出标识消息长度的字段偏移量和消息长度自身所占的字节数，Netty就能自动实现对半包的处理。对于业务解码器来说，调用父类LengthFieldBased FrameDecoder的解码方法后，返回的就是整包消息或者为空，如果为空说明是个半包消息，直接返回继续由I/O线程读取后续的码流，代码如图14-3所示。

图14-3 半包解码代码

代码清单14-6 Netty消息解码工具类：MarshallingDecoder

```
1. public class MarshallingDecoder {
2.     private final Unmarshaller unmarshaller;
3.
4.     /**
5.      * Creates a new decoder whose maximum object size is .
6.      * If the size of the received object is greater than .
7.      * a {@link StreamCorruptedException} will be raised.
8.      *
```

```

9.      * @throws IOException
10.      *
11.      */
12.      public MarshallingDecoder() throws IOException {
13.  unmarshaller = MarshallingCodecFactory.buildUnMarshalling
14.      }
15.
16.      protected Object decode(ByteBuf in) throws Exception {
17.  int objectSize = in.readInt();
18.  ByteBuf buf = in.slice(in.readerIndex(), objectSize);
19.  ByteInput input = new ChannelBufferByteInput(buf);
20.  try {
21.      unmarshaller.start(input);
22.      Object obj = unmarshaller.readObject();
23.      unmarshaller.finish();
24.      in.readerIndex(in.readerIndex() + objectSize);
25.      return obj;
26.  } finally {
27.      unmarshaller.close();
28.  }
29.  }
30. }

```

消息的编解码类按照14.2.6章节的消息编解码模块设计实现即可，如果读者对二进制编解码比较熟悉，结合第9章对JBoss Marshall序列化框架的介绍，相信可以比较轻松地读懂本章节的代码。如果对本章节的

代码阅读起来比较吃力，建议补充下JDK的ByteBuffer和Jboss Marshall框架的相关知识，然后再学习本章。

14.3.3 握手和安全认证

握手的发起是在客户端和服务端TCP链路建立成功通道激活时，握手消息的接入和安全认证在服务端处理。下面看下具体实现。

首先开发一个握手认证的客户端ChannelHandler，用于在通道激活时发起握手请求，具体代码实现如下。

代码清单14-7 LoginAuthReqHandler

```
1. public class LoginAuthReqHandler extends ChannelHandlerAdapter {
2.
3.     /**
4.      * Calls {@link ChannelHandlerContext#fireChannelActive} to
5.      * next {@link ChannelHandler} in the {@link ChannelPipeline}
6.      *
7.      * Sub-classes may override this method to change behavior
8.      */
9.     @Override
10.    public void channelActive(ChannelHandlerContext ctx) {
11.        ctx.writeAndFlush(buildLoginReq());
12.    }
13.
14.    /**
15.     * Calls {@link ChannelHandlerContext#fireChannelRead} to
```

```

16.      * the next {@link ChannelHandler} in the {@link ChannelHandlerContext}
17.      *
18.      * Sub-classes may override this method to change behavior
19.      */
20.      @Override
21.      public void channelRead(ChannelHandlerContext ctx, Object msg)
22.      throws Exception {
23.      NettyMessage message = (NettyMessage) msg;
24.
25.      // 如果是握手应答消息，需要判断是否认证成功
26.      if (message.getHeader() != null
27.          && message.getHeader().getType() == MessageType.LOGIN_RESP
28.              .value()) {
29.          byte loginResult = (byte) message.getBody();
30.          if (loginResult != (byte) 0) {
31.              // 握手失败，关闭连接
32.              ctx.close();
33.          } else {
34.              System.out.println("Login is ok : " + message);
35.              ctx.fireChannelRead(msg);
36.          }
37.      } else {
38.          ctx.fireChannelRead(msg);
39.      }
40.
41.      private NettyMessage buildLoginReq() {
42.      NettyMessage message = new NettyMessage();

```

```

43. Header header = new Header();
44. header.setType(MessageType.LOGIN_REQ.value());
45. message.setHeader(header);
46. return message;
47.     }
48.
49.     public void exceptionCaught(ChannelHandlerContext ctx,
50.         throws Exception {
51. ctx.fireExceptionCaught(cause);
52.     }
53. }

```

第10~12行，当客户端跟服务端TCP三次握手成功之后，由客户端构造握手请求消息发送给服务端，由于采用IP白名单认证机制，因此，不需要携带消息体，消息体为空，消息类型为3：握手请求消息。握手请求发送之后，按照协议规范，服务端需要返回握手应答消息。

第21~39行对握手应答消息进行处理，首先判断消息是否是握手应答消息，如果不是，直接透传给后面的ChannelHandler进行处理；如果是握手应答消息，则对应答结果进行判断，如果非0，说明认证失败，关闭链路，重新发起连接。

接着看服务端的握手接入和安全认证代码。

代码清单14-8 LoginAuthRespHandler

```

1. public class LoginAuthRespHandler extends ChannelHandlerAd

```

```

2.     private Map<String, Boolean> nodeCheck = new ConcurrentMap<>();
3.     private String[] whiteList = { "127.0.0.1", "192.168.1.1" };
4.
5.     /**
6.      * Calls {@link ChannelHandlerContext#fireChannelRead(Object)} to
7.      * the next {@link ChannelHandler} in the {@link ChannelHandlerContext}
8.      *
9.      * Sub-classes may override this method to change behavior.
10.    */
11.    @Override
12.    public void channelRead(ChannelHandlerContext ctx, Object msg)
13.        throws Exception {
14.        NettyMessage message = (NettyMessage) msg;
15.
16.        // 如果是握手请求消息，处理，其他消息透传
17.        if (message.getHeader() != null
18.            && message.getHeader().getType() == MessageType.LOGIN_
19.                .value()) {
20.            String nodeIndex = ctx.channel().remoteAddress().toString();
21.            NettyMessage loginResp = null;
22.            // 重复登录，拒绝
23.            if (nodeCheck.containsKey(nodeIndex)) {
24.                loginResp = buildResponse((byte) -1);
25.            } else {
26.                InetSocketAddress address = (InetSocketAddress) ctx.channel()
27.                    .remoteAddress();
28.                String ip = address.getAddress().getHostAddress();

```

```

29.     boolean isOK = false;
30.     for (String WIP : whitekList) {
31.         if (WIP.equals(ip)) {
32.             isOK = true;
33.             break;
34.         }
35.     }
36.     loginResp = isOK ? buildResponse((byte) 0)
37.         : buildResponse((byte) -1);
38.     if (isOK)
39.         nodeCheck.put(nodeIndex, true);
40.     }
41.     System.out.println("The login response is : " + loginR
42.         + " body [" + loginResp.getBody() + "]);
43.     ctx.writeAndFlush(loginResp);
44. } else {
45.     ctx.fireChannelRead(msg);
46. }
47. }
48.
49.     private NettyMessage buildResponse(byte result) {
50. NettyMessage message = new NettyMessage();
51. Header header = new Header();
52. header.setType(MessageType.LOGIN_RESP.value());
53. message.setHeader(header);
54. message.setBody(result);
55. return message;

```



```
56.     }
57.
58.     public void exceptionCaught(ChannelHandlerContext ctx,
59.         throws Exception {
60. nodeCheck.remove(ctx.channel().remoteAddress().toString());
61. ctx.close();
62. ctx.fireExceptionCaught(cause);
63.     }
64. }
```

第2、3行分别定义了重复登录保护和IP认证白名单列表，主要用于提升握手的可靠性。第17~47行用于接入认证，首先根据客户端的源地址（/127.0.0.1:12088）进行重复登录判断，如果客户端已经登录成功，拒绝重复登录，以防止由于客户端重复登录导致的句柄泄漏。随后通过ChannelHandlerContext的Channel接口获取客户端的InetSocketAddress地址，从中取得发送方的源地址信息，通过源地址进行白名单校验，校验通过握手成功，否则握手失败。最后通过buildResponse构造握手应答消息返回给客户端。

当发生异常关闭链路的时候，需要将客户端的信息从登录注册表中去注册，以保证后续客户端可以重连成功。

14.3.4 心跳检测机制

握手成功之后，由客户端主动发送心跳消息，服务端接收到心跳消息之后，返回心跳应答消息。由于心跳消息的目的是为了检测链路的可用性，因此不需要携带消息体。

客户端发送心跳请求消息的代码如下。

代码清单14-9 HeartBeatReqHandler

```
1. public class HeartBeatReqHandler extends ChannelHandlerAdapter {
2.     private volatile ScheduledFuture<?> heartBeat;
3.
4.     @Override
5.     public void channelRead(ChannelHandlerContext ctx, Object msg)
6.         throws Exception {
7.         NettyMessage message = (NettyMessage) msg;
8.         // 握手成功，主动发送心跳消息
9.         if (message.getHeader() != null
10.            && message.getHeader().getType() == MessageType.LOGIN
11.                .value()) {
12.             heartBeat = ctx.executor().scheduleAtFixedRate(
13.                 new HeartBeatReqHandler.HeartBeatTask(ctx), 0, 5000,
14.                 TimeUnit.MILLISECONDS);
15.         } else if (message.getHeader() != null
16.            && message.getHeader().getType() == MessageType.HEARTBEAT
17.                .value()) {
18.             System.out
19.                 .println("Client receive server heart beat message: "
20.                     + message);
21.         } else
22.             ctx.fireChannelRead(msg);
23.     }
```

```
24.
25.     private class HeartBeatTask implements Runnable {
26. private final ChannelHandlerContext ctx;
27.
28. public HeartBeatTask(final ChannelHandlerContext ctx) {
29.     this.ctx = ctx;
30. }
31.
32. @Override
33. public void run() {
34.     NettyMessage heatBeat = buildHeatBeat();
35.     System.out
36.         .println("Client send heart beat message to serv
37.             + heatBeat);
38.     ctx.writeAndFlush(heatBeat);
39. }
40.
41. private NettyMessage buildHeatBeat() {
42.     NettyMessage message = new NettyMessage();
43.     Header header = new Header();
44.     header.setType(MessageType.HEARTBEAT_REQ.value());
45.     message.setHeader(header);
46.     return message;
47. }
48. }
49.
50. @Override
```

```
51.     public void exceptionCaught(ChannelHandlerContext ctx)
52.     throws Exception {
53.     if (heartBeat != null) {
54.         heartBeat.cancel(true);
55.         heartBeat = null;
56.     }
57.     ctx.fireExceptionCaught(cause);
58. }
59. }
```

首先看第9行，当握手成功之后，握手请求Handler会继续将握手成功消息向下透传，HeartBeatReqHandler接收到之后对消息进行判断，如果是握手成功消息，则启动无限循环定时器用于定期发送心跳消息。由于NioEventLoop是一个schedule，因此它支持定时器的执行。心跳定时器的单位是毫秒，默认为5000，即每5秒发送一条心跳消息。

为了统一在一个handler中处理所有的心跳消息，因此第15~20行用于接收服务端发送的心跳应答消息，并打印客户端接收和发送的心跳消息。

心跳定时器HeartBeatTask的实现很简单，通过构造函数获取ChannelHandlerContext，构造心跳消息并发送。

服务端的心跳应答handler代码如下。

代码清单14-10 HeartBeatRespHandler

```
1. public class HeartBeatRespHandler extends ChannelHandlerAdaptor {
```

```

2.     @Override
3.     public void channelRead(ChannelHandlerContext ctx, Object msg)
4.         throws Exception {
5.         NettyMessage message = (NettyMessage) msg;
6.         // 返回心跳应答消息
7.         if (message.getHeader() != null
8.             && message.getHeader().getType() == MessageType.HEARTBEAT_RESP
9.                 .value()) {
10.            System.out.println("Receive client heart beat message: "
11.                                + message);
12.            NettyMessage heartBeat = buildHeartBeat();
13.            System.out
14.                .println("Send heart beat response message to client: "
15.                            + heartBeat);
16.            ctx.writeAndFlush(heartBeat);
17.        } else
18.            ctx.fireChannelRead(msg);
19.        }
20.
21.        private NettyMessage buildHeartBeat() {
22.            NettyMessage message = new NettyMessage();
23.            Header header = new Header();
24.            header.setType(MessageType.HEARTBEAT_RESP.value());
25.            message.setHeader(header);
26.            return message;
27.        }
28.    }

```

服务端的心跳Handler非常简单，接收到心跳请求消息之后，构造心跳应答消息返回，并打印接收和发送的心跳消息。

心跳超时的实现非常简单，直接利用Netty的ReadTimeoutHandler机制，当一定周期内（默认值50s）没有读取到对方任何消息时，需要主动关闭链路。如果是客户端，重新发起连接；如果是服务端，释放资源，清除客户端登录缓存信息，等待服务端重连。

具体代码实现在下面的小节中会进行说明。

14.3.5 断连重连

当客户端感知断连事件之后，释放资源，重新发起连接，具体代码实现如图14-4所示。

图14-4 客户端重连代码

首先监听网络断连事件，如果Channel关闭，则执行后续的重连任务，通过Bootstrap重新发起连接，客户端挂在closeFuture上监听链路关闭信号，一旦关闭，则创建重连定时器，5s之后重新发起连接，直到重连成功。

服务端感知到断连事件之后，需要清空缓存的登录认证注册信息，以保证后续客户端能够正常重连。

14.3.6 客户端代码

客户端主要用于初始化系统资源，根据配置信息发起连接，代码如

下。

代码清单14-11 NettyClient

```
1. public class NettyClient {
2.     private ScheduledExecutorService executor = Executors
3.         .newScheduledThreadPool(1);
4.     EventLoopGroup group = new NioEventLoopGroup();
5.     public void connect(int port, String host) throws Excep
6.     // 配置客户端NIO线程组
7.     try {
8.         Bootstrap b = new Bootstrap();
9.         b.group(group).channel(NioSocketChannel.class)
10.            .option(ChannelOption.TCP_NODELAY, true)
11.            .handler(new ChannelInitializer<SocketChannel>() {
12.                @Override
13.                public void initChannel(SocketChannel ch)
14.                    throws Exception {
15.                    ch.pipeline().addLast(
16.                        new NettyMessageDecoder(1024 * 1024, 4, 4)
17.                    ch.pipeline().addLast("MessageEncoder",
18.                        new NettyMessageEncoder());
19.                    ch.pipeline().addLast("readTimeoutHandler",
20.                        new ReadTimeoutHandler(50));
21.                    ch.pipeline().addLast("LoginAuthHandler",
22.                        new LoginAuthReqHandler());
23.                    ch.pipeline().addLast("HeartBeatHandler",
```

```
24.             new HeartBeatReqHandler());
25.         }
26.     });
27.     // 发起异步连接操作
28.     ChannelFuture future = b.connect(
29.         new InetSocketAddress(host, port),
30.         new InetSocketAddress(NettyConstant.LOCALIP,
31.             NettyConstant.LOCAL_PORT)).sync();
32.     future.channel().closeFuture().sync();
33. } finally {
34.     // 所有资源释放完成之后，清空资源，再次发起重连操作
35.     executor.execute(new Runnable() {
36.         @Override
37.         public void run() {
38.             try {
39.                 TimeUnit.SECONDS.sleep(5);
40.                 try {
41.                     connect(NettyConstant.PORT, NettyConstant.REMO
42.                 } catch (Exception e) {
43.                     e.printStackTrace();
44.                 }
45.                 } catch (InterruptedException e) {
46.                     e.printStackTrace();
47.                 }
48.             }
49.         });
50. }
```



```
51.     }
52.
53.     /**
54.      * @param args
55.      * @throws Exception
56.      */
57.     public static void main(String[] args) throws Exception
58.     new NettyClient().connect(NettyConstant.PORT, NettyConstant
59.     }
60. }
```

第15和16行增加了NettyMessageDecoder用于Netty消息解码，为了防止由于单条消息过大导致的内存溢出或者畸形码流导致解码错位引起内存分配失败，我们对单条消息最大长度进行了上限限制。第17和18行新增了Netty消息编码器，用于协议消息的自动编码。随后依次增加了读超时Handler、握手请求Handler和心跳消息Handler。

第28行发起TCP连接的代码与之前的不同，这次我们绑定了本地端口，主要用于服务端重复登录保护，另外，从产品管理角度看，一般情况下不允许系统随便使用随机端口。

利用Netty的ChannelPipeline和ChannelHandler机制，可以非常方便地实现功能解耦和业务产品的定制。例如本例程中的心跳定时器、握手请求和后端的业务处理可以通过不同的Handler来实现，类似于AOP。通过Handler Chain的机制可以方便地实现切面拦截和定制，相比于AOP它的性能更高。

14.3.7 服务端代码

相对于客户端，服务端的代码更简单一些，主要的工作就是握手的接入认证等，不用关心断连重连等事件。

服务端的代码如下。

代码清单14-12 NettyServer

```
1. public class NettyServer {
2.     public void bind() throws Exception {
3.         // 配置服务端的NIO线程组
4.         EventLoopGroup bossGroup = new NioEventLoopGroup();
5.         EventLoopGroup workerGroup = new NioEventLoopGroup();
6.         ServerBootstrap b = new ServerBootstrap();
7.         b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
8.           .option(ChannelOption.SO_BACKLOG, 100)
9.           .handler(new LoggingHandler(LogLevel.INFO))
10.          .childHandler(new ChannelInitializer<SocketChannel>() {
11.              @Override
12.              public void initChannel(SocketChannel ch)
13.                  throws IOException {
14.                  ch.pipeline().addLast(
15.                      new NettyMessageDecoder(1024 * 1024, 4, 4));
16.                  ch.pipeline().addLast(new NettyMessageEncoder());
17.                  ch.pipeline().addLast("readTimeoutHandler",
18.                      new ReadTimeoutHandler(50));
```

```

19.         ch.pipeline().addLast(new LoginAuthRespHandler(.
20.         ch.pipeline().addLast("HeartBeatHandler",
21.             new HeartBeatRespHandler());
22.     }
23. });
24.
25. // 绑定端口，同步等待成功
26. b.bind(NettyConstant.REMOTEIP, NettyConstant.PORT).sync(),
27. System.out.println("Netty server start ok : "
28.     + (NettyConstant.REMOTEIP + " : " + NettyConstant.PORT
29.     }
30.
31.     public static void main(String[] args) throws Exceptio
32. new NettyServer().bind();
33.     }
34. }

```

与客户端不同的是，服务端ChannelPipeline中除了Netty编码器和解码器以外，还有握手和接入认证的LoginAuthRespHandler和心跳应答HeartBeatRespHandler。

14.4 运行协议栈

14.4.1 正常场景

启动服务端，待服务端启动成功之后启动客户端，检查链路是否建立成功，是否每隔5s互发一次心跳请求和应答，运行结果如图14-5所示。

图14-5 服务端运行结果

客户端运行结果如图14-6所示。

图14-6 客户端运行结果

从上面的运行结果可以看出，客户端和服务端握手成功，双方可以互发心跳，链路正常，如图14-7所示。

图14-7 TCP链接正常

14.4.2 异常场景：服务端宕机重启

假设服务端宕机一段时间重启，检验如下功能是否正常。

- (1) 客户端是否能够正常发起重连；
- (2) 重连成功之后，不再重连；
- (3) 断连期间，心跳定时器停止工作，不再发送心跳请求消息；
- (4) 服务端重启成功之后，允许客户端重新登录；

(5) 服务端重启成功之后，客户端能够重连和握手成功；

(6) 重连成功之后，双方的心跳能够正常互发。

(7) 性能指标：重连期间，客户端资源得到了正常回收，不会导致句柄等资源泄漏。

服务端重启之前的客户端资源占用如图14-8所示。

图14-8 客户端堆内存占用

线程资源占用如图14-9所示。

图14-9 客户端线程资源信息列表

服务端宕机之后，重启之前，客户端周期性重连失败。如图14-10所示。

图14-10 客户端重连失败

重连期间线程资源占用正常，如图14-11所示。

图14-11 重连期间线程资源占用正常

重连期间内存占用正常，如图14-12所示。

图14-12 重连期间内存占用正常

服务端重启成功，握手成功，链路重新恢复，如图14-13所示。

图14-13 服务端重启成功后链路恢复

通过netstat命令查看TCP连接状态，如图14-14所示。

图14-14 TCP连接正常

14.4.3 异常场景：客户端宕机重启

客户端宕机重启之后，服务端需要能够清除缓存信息，允许客户端重新登录。下面看测试结果。

客户端停机，然后重启，结果如图14-15所示。

图14-15 客户端宕机重启重新登录

运行结果表明客户端重启之后可以重新登录成功，说明服务端功能正常。

14.5 总结

本章首先介绍了私有协议栈的相关概念，然后通过一个模拟私有协议栈——Netty协议栈的设计和开发，让读者掌握私有协议栈的功能和开发要点，为后续在实际工作中进行私有协议栈的设计和开发提供帮助。

尽管本章节在设计Netty协议栈的时候，已经考虑了很多可靠性方面的功能，但是对于实际商用协议栈而言，仍然是不足的。例如当链路断连的时候，已经放入发送队列中的消息不能丢失，更加通用的做法是提供通知机制，将发送失败的消息通知给业务侧，由业务做决定：是丢弃还是缓存重发。

本章综合了之前所学的Netty知识，还涉及到了通用半包解码器、读超时、自定义定时任务、安全认证等方面的知识，当读者能够综合运用所学知识进行灵活设计和开发时，说明对Netty的掌握程度更上了一层楼。

需要指出的是，本例程仅仅是个简单Demo，限于篇幅，一些实现未必是最优的，读者在学习过程中也可以思考下哪些地方还可以进一步优化。

源码分析篇 **Netty**功能介绍和源码分析

第15章 ByteBuf和相关辅助类

第16章 Channel和Unsafe

第17章 ChannelPipeline和ChannelHandler

第18章 EventLoop和EventLoopGroup

第19章 Future和Promise

第15章 ByteBuf和相关辅助类

从本章开始，我们将学习Netty NIO相关的主要接口和模块的API功能，并对其源码实现进行分析，希望读者通过对功能和API的学习，能够更加熟练地掌握和应用这些类库。对源码的学习不仅能够帮助读者从源码的层面掌握Netty框架，方便日后的维护、扩展和定制，更能够起到触类旁通的作用，拓展读者的知识面，提升编程技能。

本章主要包括：

- ByteBuf 功能说明
- ByteBuf源码分析
- ByteBuf相关辅助类功能说明

15.1 ByteBuf功能说明

当我们进行数据传输的时候，往往需要使用到缓冲区，常用的缓冲区就是JDK NIO类库提供的java.nio.Buffer，它的实现类如图15-1所示。

图15-1 java.nio.Buffer继承关系图

实际上，7种基础类型（Boolean除外）都有自己的缓冲区实现，对于NIO编程而言，我们主要使用的是ByteBuffer。从功能角度而言，ByteBuffer完全可以满足NIO编程的需要，但是由于NIO编程的复杂性，ByteBuffer也有其局限性，它的主要缺点如下。

（1）ByteBuffer长度固定，一旦分配完成，它的容量不能动态扩展和收缩，当需要编码的POJO对象大于ByteBuffer的容量时，会发生索引越界异常；

（2）ByteBuffer只有一个标识位置的指针position，读写的时候需要手工调用flip()和rewind()等，使用者必须小心谨慎地处理这些API，否则很容易导致程序处理失败；

（3）ByteBuffer的API功能有限，一些高级和实用的特性它不支持，需要使用者自己编程实现。

为了弥补这些不足，Netty提供了自己的ByteBuffer实现——ByteBuf，下面我们一起学习ByteBuf的原理和主要功能。

15.1.1 ByteBuf的工作原理

不同ByteBuf实现类的工作原理不尽相同，本小节我们从ByteBuf的

设计原理出发，一起探寻Netty ByteBuf的设计理念。

首先，ByteBuf依然是个Byte数组的缓冲区，它的基本功能应该与JDK的ByteBuffer一致，提供以下几类基本功能。

- 7种Java基础类型、byte数组、ByteBuffer（ByteBuf）等的读写；
- 缓冲区自身的copy和slice等；
- 设置网络字节序；
- 构造缓冲区实例；
- 操作位置指针等方法。

由于JDK的ByteBuffer已经提供了这些基础能力的实现，因此，Netty ByteBuf的实现可以有两种策略。

- 参考JDK ByteBuffer的实现，增加额外的功能，解决原ByteBuffer的缺点；
- 聚合JDK ByteBuffer，通过Facade模式对其进行包装，可以减少自身的代码量，降低实现成本。

JDK ByteBuffer由于只有一个位置指针用于处理读写操作，因此每次读写的时候都需要额外调用flip()和clear()等方法，否则功能将出错，它的典型用法如下。

```
ByteBuffer buffer = ByteBuffer.allocate(88);  
String value = "Netty权威指南";  
buffer.put(value.getBytes());  
buffer.flip();  
byte
```

```
[ ] vArray = new byte

[buffer.remaining()];
    buffer.get(vArray);
    String decodeValue = new

String(vArray);
```

我们看下调用flip()操作前后的对比。

图15-2 ByteBuffer flip()操作之前

如图15-2所示，如果不做flip操作，读取到的将是position到capacity之间的错误内容。

当执行flip()操作之后，它的limit被设置为position，position设置为0，capacity不变，由于读取的内容是从position到limit之间，因此，它能够正确地读取到之前写入缓冲区的内容。如图15-3所示。

图15-3 ByteBuffer flip()操作之后

ByteBuf通过两个位置指针来协助缓冲区的读写操作，读操作使用readerIndex，写操作使用writerIndex。

readerIndex和writerIndex的取值一开始都是0，随着数据的写入

writerIndex会增加，读取数据会使readerIndex增加，但是它不会超过writerIndex。在读取之后，0~readerIndex的就被视为discard的，调用discardReadBytes方法，可以释放这部分空间，它的作用类似ByteBuffer的compact方法。ReaderIndex和writerIndex之间的数据是可读取的，等价于ByteBuffer position和limit之间的数据。WriterIndex和capacity之间的空间是可写的，等价于ByteBuffer limit和capacity之间的可用空间。

由于写操作不修改readerIndex指针，读操作不修改writerIndex指针，因此读写之间不再需要调整位置指针，这极大地简化了缓冲区的读写操作，避免了由于遗漏或者不熟悉flip()操作导致的功能异常。

初始分配的ByteBuf如图15-4所示。

图15-4 初始分配的ByteBuf

写入 N 个字节之后的ByteBuf如图15-5所示。

图15-5 写入 N 个字节后的ByteBuf

读取 M ($< N$) 个字节之后的ByteBuf如图15-6所示。

图15-6 读取 M 个字节后的ByteBuf

调用discardReadBytes操作之后的ByteBuf如图15-7所示。

图15-7 discardReadBytes操作之后的ByteBuf

调用clear操作之后的ByteBuf如图15-8所示。

图15-8 clear操作之后的ByteBuf

下面我们继续分析ByteBuf是如何实现动态扩展的。通常情况下，当我们对ByteBuffer进行put操作的时候，如果缓冲区剩余可写空间不够，就会发生BufferOverflowException异常。为了避免发生这个问题，通常在进行put操作的时候会对剩余可用空间进行校验，如果剩余空间不足，需要重新创建一个新的ByteBuffer，并将之前的ByteBuffer复制到新创建的ByteBuffer中，最后释放老的ByteBuffer，代码示例如下。

```
        if
            (this.

buffer.remaining() < needSize)
        {
            int

toBeExtSize = needSize < 128 ? needSize : 128;
            ByteBuffer tmpBuffer = ByteBuffer.allocate(this.buffer
            this.

buffer.flip();
            tmpBuffer.put(this.
```

```
buffer);  
    this.
```

```
buffer = tmpBuffer;  
    }
```

从示例代码可以看出，为了防止ByteBuffer溢出，每进行一次put操作，都需要对可用空间进行校验，这导致了代码冗余，稍有不慎，就可能引入其他问题。为了解决这个问题，ByteBuffer对write操作进行了封装，由ByteBuffer的write操作负责进行剩余可用空间的校验，如果可用缓冲区不足，ByteBuffer会自动进行动态扩展，对于使用者而言，不需要关心底层的校验和扩展细节，只要不超过设置的最大缓冲区容量即可。当可用空间不足时，ByteBuffer会帮助我们实现自动扩展，这极大地降低了ByteBuffer的学习和使用成本，提升了开发效率。校验和扩展的相关代码如图15-9、15-10所示。

图15-9 ByteBuffer写入字节

图15-10 ByteBuffer写入字节

通过源码分析，我们发现当进行write操作时会对需要write的字节进行校验，如果可写的字节数小于需要写入的字节数，并且需要写入的字节数小于可写的最大字节数时，对缓冲区进行动态扩展。无论缓冲区是否进行了动态扩展，从功能角度看使用者并不感知，这样就简化了上层的应用。

由于NIO的Channel读写的参数都是ByteBuffer，因此，Netty的ByteBuf接口必须提供API方便的将ByteBuf转换成ByteBuffer，或者将ByteBuffer包装成ByteBuf。考虑到性能，应该尽量避免缓冲区的复制，内部实现的时候可以考虑聚合一个ByteBuffer的私有指针用来代表ByteBuffer。在后面的源码分析章节我们将详细介绍它的实现原理。

学习完ByteBuf的原理之后，下面我们继续学习它的主要API功能。

15.1.2 ByteBuf的功能介绍

本小节我们将对ByteBuf的常用API进行分类说明，讲解它的主要功能，后面的章节将给出重要API的一些典型用法。

1. 顺序读操作（**read**）

ByteBuf的read操作类似于ByteBuffer的get操作，主要的API功能说明如表15-1所示。

表15-1 ByteBuf的读操作API列表

2. 顺序写操作（**write**）

ByteBuf的write操作类似于ByteBuffer的put操作，主要的API功能说明如表15-2所示。

表15-2 ByteBuf的写操作API列表

3. **readerIndex**和**writerIndex**

Netty提供了两个指针变量用于支持顺序读取和写入操作：
`readerIndex`用于标识读取索引，`writerIndex`用于标识写入索引。两个位置指针将`ByteBuf`缓冲区分割成三个区域，如图15-11所示。

图15-11 `ByteBuf`的`readerIndex`和`writerIndex`

调用`ByteBuf`的`read`操作时，从`readerIndex`处开始读取。`readerIndex`到`writerIndex`之间的空间为可读的字节缓冲区；从`writerIndex`到`capacity`之间为可写的字节缓冲区；0到`readerIndex`之间是已经读取过的缓冲区，可以调用`discardReadBytes`操作来重用这部分空间，以节约内存，防止`ByteBuf`的动态扩张。这在私有协议栈消息解码的时候非常有用，因为TCP底层可能粘包，几百个整包消息被TCP粘包后作为一个整包发送，这样，通过`discardReadBytes`操作可以重用之前已经解码过的缓冲区，这样就可以防止接收缓冲区因为容量不足导致的扩张。但是，`discardReadBytes`操作是把双刃剑，不能滥用，关于这一点在后续章节会进行详细说明。

4. Discardable bytes

相比于其他的Java对象，缓冲区的分配和释放是个耗时的操作，因此，我们需要尽量重用它们。由于缓冲区的动态扩张需要进行字节数组的复制，它是个耗时的操作，因此，为了最大程度地提升性能，往往需要尽最大努力提升缓冲区的重用率。

假如缓冲区包含了 N 个整包消息，每个消息的长度为 L ，消息的可写字节数为 R 。当读取 M 个整包消息后，如果不对`ByteBuf`做压缩或者`discardReadBytes`操作，则可写的缓冲区长度依然为 R 。如果调用`discardReadBytes`操作，则可写字节数会变为 $R = (R + M \times L)$ ，之前已

经读取的 M 个整包的空间会被重用。假如此时ByteBuf需要写入 $R + 1$ 个字节，则不需要动态扩张ByteBuf。

ByteBuf的discardReadBytes操作效果图如下。

操作之前如图15-12所示。

图15-12 discardReadBytes操作之前的ByteBuf

操作之后如图15-13所示。

图15-13 discardReadBytes操作之后的ByteBuf

需要指出的是，调用discardReadBytes会发生字节数组的内存复制，所以，频繁调用将会导致性能下降，因此在调用它之前要确认你确实需要这样做，例如牺牲性能来换取更多的可用内存。

discardReadBytes的相关源码如图15-14所示。

图15-14 discardReadBytes操作会导致内存复制

需要指出的是，调用discardReadBytes操作之后的writable bytes内容处理策略跟ByteBuf接口的具体实现有关。

5. Readable bytes和Writable bytes

可读空间段是数据实际存储的区域，以read或者skip开头的任何操作将会从readerIndex开始读取或者跳过指定的数据，操作完成之后readerIndex增加了读取或者跳过的字节数长度。如果读取的字节数长度大于实际可读的字节数，则抛出IndexOutOfBoundsException。当新分配、包装或者复制一个新的ByteBuf对象时，它的readerIndex为0。

可写空间段是尚未被使用可以填充的空闲空间，任何以write开头的操作都会从writerIndex开始向空闲空间写入字节，操作完成之后writerIndex增加了写入的字节数长度。如果写入的字节数大于可写的字节数，则会抛出IndexOutOfBoundsException异常。新分配一个ByteBuf对象时，它的readerIndex为0。通过包装或者复制的方式创建一个新的ByteBuf对象时，它的writerIndex是ByteBuf的容量。

6. Clear操作

正如JDK ByteBuffer的clear操作，它并不会清空缓冲区内容本身，例如填充为NUL（0x00）。它主要用来操作位置指针，例如position、limit和mark。对于ByteBuf，它也是用来操作readerIndex和writerIndex，将它们还原为初始分配值。具体的处理示例图如下。

Clear()操作之前如图15-15所示。

图15-15 clear操作之前的缓冲区

Clear() 操作之后如图15-16所示。

图15-16 clear操作之后的缓冲区

7. Mark和Rest

当对缓冲区进行读操作时，由于某种原因，可能需要对之前的操作进行回滚。读操作并不会改变缓冲区的内容，回滚操作主要就是重新设置索引信息。

对于JDK的ByteBuffer，调用mark操作会将当前的位置指针备份到

mark变量中，当调用rest操作之后，重新将指针的当前位置恢复为备份在mark中的值，源码代码如图15-17所示。

图15-17 mark操作之后的缓冲区位置指针

调用reset操作之后如图15-18所示。

图15-18 rest操作之后的缓冲区位置指针

Netty的ByteBuf也有类似的rest和mark接口，因为ByteBuf有读索引和写索引，因此，它总共有4个相关的方法，分别如下。

- markReaderIndex：将当前的readerIndex备份到markedReaderIndex中；
- resetReaderIndex：将当前的readerIndex设置为markedReaderIndex；
- markWriterIndex：将当前的writerIndex备份到markedWriterIndex；
- resetWriterIndex：将当前的writerIndex设置为markedWriterIndex。

相关的代码实现如图15-19所示。

图15-19 mark和rest操作前后的缓冲区位置指针

8. 查找操作

很多时候，需要从ByteBuf中查找某个字符，例如通过“\r\n”作为文本字符串的换行符，利用“NUL(0x00)”作为分隔符。

ByteBuf提供了多种查找方法用于满足不同的应用场景，详细分类如下。

(1) `indexOf(int fromIndex, int toIndex, byte value)`: 从当前ByteBuf中定位出首次出现value的位置, 起始索引为fromIndex, 终点是toIndex, 如果没有查找到则返回-1, 否则返回第一条满足搜索条件的位置索引。

(2) `bytesBefore(byte value)`: 从当前ByteBuf中定位出首次出现value的位置, 起始索引为readerIndex, 终点是writerIndex, 如果没有查找到则返回-1, 否则返回第一条满足搜索条件的位置索引。该方法不会修改readerIndex和writerIndex。

(3) `bytesBefore(int length, byte value)`: 从当前ByteBuf中定位出首次出现value的位置, 起始索引为readerIndex, 终点是readerIndex+length, 如果没有查找到则返回-1, 否则返回第一条满足搜索条件的位置索引。如果length大于当前字节缓冲区的可读字节数, 则抛出IndexOutOfBoundsException异常。

(4) `bytesBefore(int index, int length, byte value)`: 从当前ByteBuf中定位出首次出现value的位置, 起始索引为index, 终点是index+length, 如果没有查找到则返回-1, 否则返回第一条满足搜索条件的位置索引。如果index+length大于当前字节缓冲区的容量, 则抛出IndexOutOfBoundsException异常。

(5) `forEachByte(ByteBufProcessor processor)`: 遍历当前ByteBuf的可读字节数组, 与ByteBufProcessor设置的查找条件进行对比, 如果满足条件, 则返回位置索引, 否则返回-1。

(6) `forEachByte(int index, int length, ByteBufProcessor processor)`: 以index为起始位置, index + length为终止位置进行遍历, 与

ByteBufProcessor设置的查找条件进行对比，如果满足条件，则返回位置索引，否则返回-1。

(7) `forEachByteDesc(ByteBufProcessor processor)`: 遍历当前ByteBuf的可读字节数组，与ByteBufProcessor设置的查找条件进行对比，如果满足条件，则返回位置索引，否则返回-1。注意对字节数组进行迭代的时候采用逆序的方式，也就是从`writerIndex-1`开始迭代，直到`readerIndex`。

(8) `forEachByteDesc(int index, int length, ByteBufProcessor processor)`: 以`index`为起始位置，`index + length`为终止位置进行遍历，与ByteBufProcessor设置的查找条件进行对比，如果满足条件，则返回位置索引，否则返回-1。采用逆序查找的方式，从`index+length-1`开始，直到`index`。

对于查找的字节而言，存在一些常用值，例如回车换行符、常用的分隔符等，Netty为了减少业务的重复定义，在ByteBufProcessor接口中对这些常用的查找字节进行了抽象，定义如下。

(1) `FIND_NUL`: `NUL (0x00)`;

(2) `FIND_CR`: `CR ('\r')`;

(3) `FIND_LF`: `LF ('\n')`;

(4) `FIND_CRLF`: `CR ('\r')`或者`LF ('\n')`;

(5) `FIND_LINEAR_WHITESPACE`: `' '`或者`\t`。

使用者也可以自定义查找规则，实现如15-20所示接口即可。

9. Derived buffers

类似于数据库的视图，ByteBuf提供了多个接口用于创建某个ByteBuf的视图或者复制ByteBuf，具体方法如下。

(1) **duplicate**: 返回当前ByteBuf的复制对象，复制后返回的ByteBuf与操作的ByteBuf共享缓冲区内容，但是维护自己独立的读写索引。当修改复制后的ByteBuf内容后，之前原ByteBuf的内容也随之改变，双方持有的是同一个内容指针引用。

(2) **copy**: 复制一个新的ByteBuf对象，它的内容和索引都是独立的，复制操作本身并不修改原ByteBuf的读写索引。

(3) **copy(int index, int length)**: 从指定的索引开始复制，复制的字节长度为length，复制后的ByteBuf内容和读写索引都与之前的独立。

(4) **slice**: 返回当前ByteBuf的可读子缓冲区，起始位置从readerIndex到writerIndex，返回后的ByteBuf与原ByteBuf共享内容，但是读写索引独立维护。该操作并不修改原ByteBuf的readerIndex和writerIndex。

(5) **slice(int index, int length)**: 返回当前ByteBuf的可读子缓冲区，起始位置从index到index+length，返回后的ByteBuf与原ByteBuf共享内容，但是读写索引独立维护。该操作并不修改原ByteBuf的readerIndex和writerIndex。

10. 转换成标准的ByteBuffer

我们知道，当通过NIO的SocketChannel进行网络读写时，操作的对象是JDK标准的java.nio.ByteBuffer，由于Netty统一使用ByteBuf替代JDK原生的java.nio.ByteBuffer，所以必须从接口层面支持两者的相互转换，下面就一起看下如何将ByteBuf转换成java.nio.ByteBuffer。

将ByteBuf转换成java.nio.ByteBuffer的方法有两个，详细说明如下。

(1) ByteBuffer.nioBuffer(): 将当前ByteBuf可读的缓冲区转换成ByteBuffer，两者共享同一个缓冲区内容引用，对ByteBuffer的读写操作并不会修改原ByteBuf的读写索引。需要指出的是，返回后的ByteBuffer无法感知原ByteBuf的动态扩展操作。

(2) ByteBuffer.nioBuffer(int index, int length): 将当前ByteBuf从index开始长度为length的缓冲区转换成ByteBuffer，两者共享同一个缓冲区内容引用，对ByteBuffer的读写操作并不会修改原ByteBuf的读写索引。需要指出的是，返回后的ByteBuffer无法感知原ByteBuf的动态扩展操作。

11. 随机读写（set和get）

除了顺序读写之外，ByteBuf还支持随机读写，它与顺序读写的最大差别在于可以随机指定读写的索引位置。

读取操作的API列表如图15-21所示。

图15-21 ByteBuf随机读操作API列表

随机写操作的API列表如图15-22所示。

图15-22 ByteBuf随机写操作API列表

无论是get还是set操作，ByteBuf都会对其索引和长度等进行合法性校验，与顺序读写一致。但是，set操作与write操作不同的是它不支持动态扩展缓冲区，所以使用者必须保证当前的缓冲区可写的字节数大于需要写入的字节长度，否则会抛出数组或者缓冲区越界异常。相关代码如图15-23所示。

图15-23 ByteBuf随机写操作不支持动态扩展缓冲区

15.2 ByteBuf源码分析

由于ByteBuf的实现非常繁杂，因此本书不会对其所有子类都进行穷举分析，我们挑选ByteBuf的主要接口实现类和主要方法进行分析说明。相信理解了这些主要功能之后，再去阅读和分析其他辅助类会更加简单。

15.2.1 ByteBuf的主要类继承关系

首先，我们通过主要功能类库的继承关系图（图15-24），来看下ByteBuf接口的不同实现。

图15-24 ByteBuf主要功能类继承关系图

从内存分配的角度看，ByteBuf可以分为两类。

（1）堆内存（HeapByteBuf）字节缓冲区：特点是内存的分配和回收速度快，可以被JVM自动回收；缺点就是如果进行Socket的I/O读写，需要额外做一次内存复制，将堆内存对应的缓冲区复制到内核Channel中，性能会有一定程度的下降。

（2）直接内存（DirectByteBuf）字节缓冲区：非堆内存，它在堆外进行内存分配，相比于堆内存，它的分配和回收速度会慢一些，但是将它写入或者从Socket Channel中读取时，由于少了一次内存复制，速度比堆内存快。

正是因为各有利弊，所以Netty提供了多种ByteBuf供开发者使用，经验表明，ByteBuf的最佳实践是在I/O通信线程的读写缓冲区使用

DirectByteBuf，后端业务消息的编解码模块使用HeapByteBuf，这样组合可以达到性能最优。

从内存回收角度看，ByteBuf也分为两类：基于对象池的ByteBuf和普通ByteBuf。两者的主要区别就是基于对象池的ByteBuf可以重用ByteBuf对象，它自己维护了一个内存池，可以循环利用创建的ByteBuf，提升内存的使用效率，降低由于高负载导致的频繁GC。测试表明使用内存池后的Netty在高负载、大并发的冲击下内存和GC更加平稳。

尽管推荐使用基于内存池的ByteBuf，但是内存池的管理和维护更加复杂，使用起来也需要更加谨慎，因此，Netty提供了灵活的策略供使用者来做选择。

下面我们对主要的功能类和方法的源码进行分析和解读，以便能够更加深刻地理解ByteBuf的实现，掌握其更加高级的功能。

15.2.2 AbstractByteBuf源码分析

AbstractByteBuf继承自ByteBuf，ByteBuf的一些公共属性和功能会在AbstractByteBuf中实现，下面我们对其属性和重要代码进行分析解读。

1. 主要成员变量

首先，像读索引、写索引、mark、最大容量等公共属性需要定义，具体定义如图15-25所示。

图15-25 AbstractByteBuf成员变量定义

我们重点关注下leakDetector，它被定义为static，意味着所有的ByteBuf实例共享同一个ResourceLeakDetector对象。ResourceLeakDetector用于检测对象是否泄漏，后面有专门章节进行讲解。

我们发现，在AbstractByteBuf中并没有定义ByteBuf的缓冲区实现，例如byte数组或者DirectByteBuffer。原因显而易见，因为AbstractByteBuf并不清楚子类到底是基于堆内存还是直接内存，因此无法提前定义。

2. 读操作簇

无论子类如何实现ByteBuf，例如UnpooledHeapByteBuf使用byte数组表示字节缓冲区，UnpooledDirectByteBuf直接使用ByteBuffer，它们的功能都是相同的，操作的结果是等价的。

因此，读操作以及其他的一些公共功能都由父类实现，差异化功能由子类实现，这也就是抽象和继承的价值所在。

read类操作的方法如图15-26所示。

图15-26 读操作方法一览表

限于篇幅，我们不能一一枚举，挑选其中框线所示的readBytes(byte[] dst, int dstIndex, int length)方法进行分析，首先看源码实现，如图15-27所示。

图15-27 readBytes(byte[] dst, int dstIndex, int length)方法源码

在读之前，首先对缓冲区的可用空间进行校验，校验的代码如图15-28所示。

图15-28 `readBytes(byte[] dst, int dstIndex, int length)`方法源码

如果读取的长度小于0，则抛出`IllegalArgumentException`异常提示参数非法；如果可写的字节数小于需要读取的长度，则抛出`IndexOutOfBoundsException`异常，由于异常中封装了详细的异常信息，所以使用者可以非常方便地进行问题定位。

校验通过之后，调用`getBytes`方法，从当前的读索引开始，复制`length`个字节到目标`byte`数组中，由于不同的子类复制操作的技术实现细节不同，因此该方法由子类实现，如图15-29所示。

图15-29 字节数组读取操作

如果读取成功，需要对读索引进行递增：`readerIndex += length`。其他类型的读取操作与之类似，不再展开介绍，感兴趣的读者可以自行阅读相关代码。

3. 写操作簇

与读取操作类似，写操作的公共行为在`AbstractByteBuf`中实现，它的API列表如图15-30所示。

图15-30 写操作方法一览

我们选择与读取配套的`writeBytes(byte[] src, int srcIndex, int length)`进行分析，它的功能是将源字节数组中从`srcIndex`开始，到`srcIndex +`

length截止的字节数组写入到当前的ByteBuf中。下面具体看源码实现，如图15-31所示。

图15-31 指定的字节数组写入缓冲区源码

首先对写入字节数组的长度进行合法性校验，校验代码如图15-32所示。

图15-32 对写入的字节数组长度进行校验

如果写入的字节数组长度小于0，则抛出IllegalArgumentExcep^{tion}异常；如果写入的字节数组长度小于当前ByteBuf可写的字节数，说明可以写入成功，直接返回；如果写入的字节数组长度大于可以动态扩展的最大可写字节数，说明缓冲区无法写入超过其最大容量的字节数组，抛出IndexOutOfBoundsException异常。

如果当前写入的字节数组长度虽然大于目前ByteBuf的可写字节数，但是通过自身的动态扩展可以满足新的写入请求，则进行动态扩展。可能有读者会产生疑问，既然需要写入的字节数组长度大于当前缓冲区可写的空间，为什么不像JDK的ByteBuffer那样抛出缓冲区越界异常呢？

在前面我们分析JDK ByteBuffer缺点的时候已经有过介绍，ByteBuffer的一个最大的缺点就是一旦完成分配之后不能动态调整其容量。由于很多场景下我们无法预先判断需要编码和解码的POJO对象长度，因此只能根据经验数据给个估计值，如果这个值偏大，就会导致内存的浪费，如果这个值偏小，遇到大消息编码的时候就会发生缓冲区溢出异常，使用者需要自己捕获这个异常，并重新计算缓冲区的大小，将原来的内容复制到新的缓冲区中，然后重置指针。这种处理策略对用户

非常不友好，而且稍有不慎，就会引入新的问题。

Netty的ByteBuffer可以动态扩展，为了保证安全性，允许使用者指定最大的容量，在容量范围内，可以先分配个较小的初始容量，后面不够用再动态扩展，这样可以达到功能和性能的最优组合。

我们继续看calculateNewCapacity方法的实现：首先需要重新计算下扩展后的容量，它有一个参数，等于writerIndex + minWritableBytes，也就是满足要求的最小容量。如图15-33所示。

图15-33 重新计算缓冲区的容量

首先设置门限阈值为4M，当需要的新容量正好等于门限阈值，则使用阈值作为新的缓冲区容量。如果新申请的内存空间大于阈值，不能采用倍增的方式（防止内存膨胀和浪费）扩张内存，采用每次步进4M的方式进行内存扩张。扩张的时候需要对扩张后的内存和最大内存（maxCapacity）进行比较，如果大于缓冲区的最大长度，则使用maxCapacity作为扩容后的缓冲区容量。

如果扩容后的新容量小于阈值，则以64为计数进行倍增，直到倍增后的结果大于或等于需要的容量值。

采用倍增或者步进算法的原因如下：如果以minNewCapacity作为目标容量，则本次扩容后的可写字节数刚好够本次写入使用。写入完成后，它的可写字节数会变为0，下次做写入操作的时候，需要再次动态扩张。这样就会形成第一次动态扩张后，每次写入操作都会进行动态扩张，由于动态扩张需要进行内存复制，频繁的内存复制会导致性能下降。

采用先倍增后步进的原因如下：当内存比较小的情况下，倍增操作并不会带来太多的内存浪费，例如64字节-->128字节-->256字节，这样的内存扩张方式对于大多数应用系统是可以接受的。但是，当内存增长到一定阈值后，再进行倍增就可能会带来额外的内存浪费，例如10M，采用倍增后变为20M，很有可能系统只需要12M，扩张到20M后会带来8M的内存浪费。由于每个客户端连接都可能维护自己独立的接收和发送缓冲区，这样随着客户读的线性增长，内存浪费也会成比例的增加，因此，达到某个阈值后就需要以步进的方式对内存进行平滑地扩张。

这个阈值是个经验值，不同的应用场景，这个值可能不同，此处，ByteBuf取值为4M。

重新计算完动态扩张后的目标容量后，需要重新创建个新的缓冲区，将原缓冲区的内容复制到新创建的ByteBuf中，最后设置读写索引和mark标签等。由于不同的子类会对应不同的复制操作，所以该方法依然是个抽象方法，由子类负责实现。如图15-34所示。

图15-34 重新分配缓冲区的容量

4. 操作索引

与索引相关的操作主要涉及设置读写索引、mark和rest等。如图15-35所示。

图15-35 索引操作相关API列表

由于这部分代码非常简单，我们就以设置读索引为例进行分析，相关代码如图15-36所示。

图15-36 重设读索引

在重新设置读索引之前需要对索引进行合法性判断，如果它小于0或者大于写索引，则抛出`IndexOutOfBoundsException`异常，设置失败。校验通过之后，将索引设置为新的值，然后返回当前的`ByteBuf`对象。

5. 重用缓冲区

前面介绍功能的时候已经简单讲解了如何通过`discardReadBytes`和`discardSomeReadBytes`方法重用已经读取过的缓冲区，下面结合`discardReadBytes`方法的实现进行分析，源码如图15-37所示。

图15-37 重用读取过的缓冲区

首先对读索引进行判断，如果为0则说明没有可重用的缓冲区，直接返回。如果读索引大于0且读索引不等于写索引，说明缓冲区中既有已经读取过的被丢弃的缓冲区，也有尚未读取的可读缓冲区。调用`setBytes(0, this, readerIndex, writerIndex - readerIndex)`方法进行字节数组复制。将尚未读取的字节数组复制到缓冲区的起始位置，然后重新设置读写索引，读索引设置为0，写索引设置为之前的写索引减去读索引（重用的缓冲区长度）。

在设置读写索引的同时，需要同时调整`markedReaderIndex`和`markedWriterIndex`，调整mark的代码如图15-38所示。

图15-38 调整mark

首先对备份的`markedReaderIndex`和需要减少的`decrement`进行判断，如果小于需要减少的值，则将`markedReaderIndex`设置为0。注意，

无论markedReaderIndex还是markedWriterIndex，它的取值都不能小于0。如果markedWriterIndex也小于需要减少的值，则markedWriterIndex置为0，否则，markedWriterIndex减去decrement之后的值就是新的markedWriterIndex。

如果需要减小的值小于markedReaderIndex，则它也一定也小于markedWriterIndex，markedReaderIndex和markedWriterIndex的新值就是减去decrement之后的取值。

如果readerIndex等于writerIndex，则说明没有可读的字节数组，那就不需要进行内存复制，直接调整mark，将读写索引设置为0即可完成缓冲区的重用，代码如图15-39所示。

图15-39 没有可读的字节数组，不需要内存复制

6. skipBytes

在解码的时候，有时候需要丢弃非法的数据报，或者跳跃过不需要读取的字节或字节数组，此时，使用skipBytes方法就非常方便。它可以忽略指定长度的字节数组，读操作时直接跳过这些数据读取后面的可读缓冲区，详细的代码实现如图15-40所示。

图15-40 跳过指定长度的缓冲区

首先判断跳过的长度是否大于当前缓冲区可读的字节数组长度，如果大于可读字节数组长度，则抛出IndexOutOfBoundsException；如果参数本身为负数，则抛出IllegalArgumentException异常。

如果校验通过，则设置新的读索引为旧的索引值与跳跃的长度之

和，然后对新的读索引进行判断，如果大于写索引，则抛出 `IndexOutOfBoundsException` 异常，如果合法，则将读索引设置为新的读索引。这样后续读操作的时候就会从新的读索引开始，跳过 `length` 个字节。

15.2.3 AbstractReferenceCountedByteBuf 源码分析

从类的名字就可以看出该类主要是对引用进行计数，类似于 JVM 内存回收的对象引用计数器，用于跟踪对象的分配和销毁，做自动内存回收。

下面通过源码来看它的具体实现。

1. 成员变量

`AbstractReferenceCountedByteBuf` 成员变量列表如图15-41所示。

图15-41 `AbstractReferenceCountedByteBuf` 成员变量列表

首先看第一个字段 `refCntUpdater`，它是 `AtomicIntegerFieldUpdater` 类型变量，通过原子的方式对成员变量进行更新等操作，以实现线程安全，消除锁。第二个字段是 `REFCNT_ FIELD_OFFSET`，它用于标识 `refCnt` 字段在 `AbstractReferenceCountedByteBuf` 中的内存地址，该内存地址的获取是 JDK 实现强相关的，如果使用 SUN 的 JDK，它通过 `sun.misc.Unsafe` 的 `objectFieldOffset` 接口来获得，`ByteBuf` 的实现子类 `UnpooledUnsafeDirectByteBuf` 和 `PooledUnsafeDirectByteBuf` 会使用到这个偏移量。

最后定义了一个 `volatile` 修饰的 `refCnt` 字段用于跟踪对象的引用次

数，使用volatile是为了解决多线程并发访问的可见性问题，此处不对volatile的用法展开说明，后续多线程章节会有详细介绍。

2. 对象引用计数器

每调用一次retain方法，引用计数器就会加一，由于可能存在多线程并发调用的场景，所以它的累加操作必须是线程安全的，下面我们一起看下它的具体实现细节，如图15-42所示。

图15-42 调用retain函数，引用计数器加一

通过自旋对引用计数器进行加一操作，由于引用计数器的初始值为1，如果申请和释放操作能够保证正确使用，则它的最小值为1，当被释放和被申请的次数相等时，就调用回收方法回收当前的ByteBuf对象。如果为0，说明对象被意外、错误地引用，抛出IllegalReferenceCountException，如果引用计数器达到整形的最大值，抛出引用越界的异常IllegalReferenceCountException，最后通过compareAndSet进行原子更新，它会使用自己获取的值跟期望值进行对比，如果期间已经被其他线程修改了，则比对失败，进行自旋，重新获取引用计数器的值再次比对，如果比对成功则对其加一。注意：compareAndSet是由操作系统层面提供的原子操作，这类原子操作被称为CAS，感兴趣的读者可以看下Java CAS的原理。

下面看下释放引用计数器的代码，如图15-43所示。

图15-43 调用释放函数，引用计数器减一

与retain方法类似，它也是在一个自旋循环里面进行判断和更新的。需要注意的是：当refCnt == 1时意味着申请和释放相等，说明对象

引用已经不可达，该对象需要被释放和垃圾回收掉，则通过调用 `deallocate` 方法来释放 `ByteBuf` 对象。

15.2.4 UnpooledHeapByteBuf源码分析

`UnpooledHeapByteBuf` 是基于堆内存进行内存分配的字节缓冲区，它没有基于对象池技术实现，这就意味着每次 I/O 的读写都会创建一个新的 `UnpooledHeapByteBuf`，频繁进行大块内存的分配和回收对性能会造成一定影响，但是相比于堆外内存的申请和释放，它的成本还是会低一些。

相比于 `PooledHeapByteBuf`，`UnpooledHeapByteBuf` 的实现原理更加简单，也不容易出现内存管理方面的问题，因此在满足性能的情况下，推荐使用 `UnpooledHeapByteBuf`。

下面我们就一起来看下 `UnpooledHeapByteBuf` 的代码实现。

1. 成员变量

首先看下 `UnpooledHeapByteBuf` 的成员变量定义，如图 15-44 所示。

图 15-44 `UnpooledHeapByteBuf` 成员变量定义

首先，它聚合了一个 `ByteBufAllocator`，用于 `UnpooledHeapByteBuf` 的内存分配，紧接着定义了一个 `byte` 数组作为缓冲区，最后定义了一个 `ByteBuffer` 类型的 `tmpNioBuf` 变量用于实现 Netty `ByteBuf` 到 JDK `NIO ByteBuffer` 的转换。

事实上，如果使用 JDK 的 `ByteBuffer` 替换 `byte` 数组也是可行的，直接

使用byte数组的根本原因就是提升性能和更加便捷地进行位操作。JDK的ByteBuffer底层实现也是byte数组，代码如图15-45所示。

图15-45 JDK ByteBuffer内部实现源码

2. 动态扩展缓冲区

在前一章介绍AbstractByteBuf的时候，我们讲到ByteBuf在最大容量范围内能够实现自动扩张，下面我们一起看下缓冲区的自动扩展在UnpooledHeapByteBuf中的实现，如图15-46所示。

图15-46 缓冲区的动态扩展

方法入口首先对新容量进行合法性校验，如果大于容量上限或者小于0，则抛出IllegalArgumentException异常。

判断新的容量值是否大于当前的缓冲区容量，如果大于则需要进行动态扩展，通过`byte[] newArray = new byte[newCapacity]`创建新的缓冲区字节数组，然后通过`System.arraycopy`进行内存复制，将旧的字节数组复制到新创建的字节数组中，最后调用`setArray`替换旧的字节数组。如图15-47所示。

图15-47 字节数组替换

需要指出的是，当动态扩容完成后，需要将原来的视图`tmpNioBuf`设置为空。

如果新的容量小于当前的缓冲区容量不需要动态扩展，但是需要截取当前缓冲区创建一个新的子缓冲区，具体的算法如下：首先判断下读

索引是否小于新的容量值，如果小于进一步判断写索引是否大于新的容量值，如果大于则将写索引设置为新的容量值（防止越界）。更新完写索引之后通过内存复制`System.arraycopy`将当前可读的字节数组复制到新创建的子缓冲区中，代码如下。

```
System.arraycopy(array, readerIndex, newArray, readerIndex, w
```

如果新的容量值小于读索引，说明没有可读的字节数组需要复制到新创建的缓冲区中，将读写索引设置为新的容量值即可。最后调用`setArray`方法替换原来的字节数组。

3. 字节数组复制

在前一章节里我们介绍`setBytes(int index, byte[] src, int srcIndex, int length)`方法的时候说它有子类实现，下面我们看看`UnpooledHeapByteBuf`如何进行字节数组的复制。如图15-48所示。

图15-48 字节数组复制

首先仍然是合法性校验，我们看下校验代码。如图15-49所示。

图15-49 字节数组复制前的校验

校验`index`和`length`的值，如果它们小于0，则抛出`IllegalArgumentException`，然后对两者之和进行判断，如果大于缓冲区的容量，则抛出`IndexOutOfBoundsException`。`srcIndex`和`srcCapacity`的校验与`index`类似，不再赘述。校验通过之后，调用`System.arraycopy(src, srcIndex, array, index, length)`方法进行字节数组的复制。

需要指出的是，ByteBuf以set和get开头读写缓冲区的方法并不会修改读写索引。

4. 转换成JDK ByteBuffer

熟悉JDK NIO ByteBuffer的读者可能会想到转换非常简单，因为ByteBuf基于byte数组实现，NIO的ByteBuffer提供了wrap方法，可以将byte数组转换成ByteBuffer对象，JDK的相关源码实现如图15-50所示。

图15-50 JDK ByteBuffer的wrap方法源码

大家的猜想是对的，下面我们一起看下UnpooledHeapByteBuf的实现，如图15-51所示。

图15-51 UnpooledHeapByteBuf的wrap源码

我们发现，唯一不同的是它还调用了ByteBuffer的slice方法，slice的功能前面已经介绍过了，此处不再展开说明。由于每次调用nioBuffer都会创建一个新的ByteBuffer，因此此处的slice方法起不到重用缓冲区内容的效果，只能保证读写索引的独立性。

5. 子类实现相关的方法

ByteBuf中的一些接口是跟具体子类实现相关的，不同的子类功能是不同的，本小节我们将列出这些不同点。

- isDirect方法：如果是基于堆内存实现的ByteBuf，它返回false，相关的代码实现如图15-52所示。

图15-52 UnpooledHeapByteBuffer的isDirect方法

- **hasArray方法**：由于UnpooledHeapByteBuffer基于字节数组实现，所以它的返回值是true。
- **array方法**：由于UnpooledHeapByteBuffer基于字节数组实现，所以它的返回值是内部的字节数组成员变量。如图15-53所示。

图15-53 UnpooledHeapByteBuffer的array方法

读者在调用array方法之前，可以先通过hasArray进行判断，如果返回false说明当前的ByteBuffer不支持array方法。

- 其他本地相关的方法有：**arrayOffset**、**hasMemoryAddress**和**memoryAddress**，这些方法的实现如图15-54所示。

图15-54 UnpooledHeapByteBuffer的地址相关方法

内存地址相关的接口主要由UnsafeByteBuffer使用，它基于SUN JDK的sun.misc.Unsafe方法实现，本书的重点并不是介绍sun.misc.Unsafe的，如果读者对sun.misc.Unsafe的实现感兴趣，可以阅读OPEN JDK的相关源码实现，也可以通过其他的DOC文档进行深入学习。

由于UnpooledDirectByteBuffer与UnpooledHeapByteBuffer的实现原理相同，不同之处就是它内部缓冲区由java.nio.DirectByteBuffer实现，当掌握了UnpooledHeapByteBuffer之后，阅读UnpooledDirectByteBuffer的代码会非常容易，所以本书不再对UnpooledDirectByteBuffer进行源码解读。

15.2.5 PooledByteBuffer内存池原理分析

由于ByteBuffer内存池的实现涉及到的类和数据结构非常多，限于篇

幅，本章节不对其源码进行展开说明，而是从设计原理角度来讲解内存池的实现。

1. PoolArena

Arena本身是指一块区域，在内存管理中，Memory Arena是指内存中的一大块连续的区域，PoolArena就是Netty的内存池实现类。

为了集中管理内存的分配和释放，同时提高分配和释放内存时候的性能，很多框架和应用都会通过预先申请一大块内存，然后通过提供相应的分配和释放接口来使用内存。这样一来，对内存的管理就被集中到几个类或者函数中，由于不再频繁使用系统调用来申请和释放内存，应用或者系统的性能也会大大提高。在这种设计思路下，预先申请的那一大块内存就被称为Memory Arena。

不同的框架，Memory Arena的实现不同，Netty的PoolArena是由多个Chunk组成的大块内存区域，而每个Chunk则由一个或者多个Page组成，因此，对内存的组织和管理也就主要集中在如何管理和组织Chunk和Page了。PoolArena中的内存Chunk定义如图15-55所示。

图15-55 Netty的Memory Arena实现

2. PoolChunk

Chunk主要用来组织和管理多个Page的内存分配和释放，在Netty中，Chunk中的Page被构建成一棵二叉树。假设一个Chunk由16个Page组成，那么这些Page将会被按照图15-56所示的形式组织起来。

图15-56 Chunk的数据结构

Page的大小是4个字节，Chunk的大小是64个字节(4×16)。整棵树有5层，第1层（也就是叶子节点所在的层）用来分配所有Page的内存，第4层用来分配2个Page的内存，依次类推。

每个节点都记录了自己在整个Memory Arena中的偏移地址，当一个节点代表的内存区域被分配出去之后，这个节点就会被标记为已分配，自这个节点以下的所有节点在后面的内存分配请求中都会被忽略。举例来说，当我们请求一个16字节的存储区域时，上面这个树中的第3层中的4个节点中的一个就会被标记为已分配，这就表示整个Memory Arena中有16个字节被分配出去了，新的分配请求只能从剩下的3个节点及其子树中寻找合适的节点。

对树的遍历采用深度优先的算法，但是在选择哪个子节点继续遍历时则是随机的，并不像通常的深度优先算法中那样总是访问左边的子节点。

3. PoolSubpage

对于小于一个Page的内存，Netty在Page中完成分配。每个Page会被切分成大小相等的多个存储块，存储块的大小由第一次申请的内存块大小决定。假如一个Page是8个字节，如果第一次申请的块大小是4个字节，那么这个Page就包含2个存储块；如果第一次申请的是8个字节，那么这个Page就被分成1个存储块。

一个Page只能用于分配与第一次申请时大小相同的内存，比如，一个4字节的Page，如果第一次分配了1字节的内存，那么后面这个Page只能继续分配1字节的内存，如果有一个申请2字节内存的请求，就需要在一个新的Page中进行分配。

Page中存储区域的使用状态通过一个long数组来维护，数组中每个long的每一位表示一个块存储区域的占用情况：0表示未占用，1表示以占用。对于一个4字节的Page来说，如果这个Page用来分配1个字节的存储区域，那么long数组中就只有一个long类型的元素，这个数值的低4位用来指示各个存储区域的占用情况。对于一个128字节的Page来说，如果这个Page也是用来分配1个字节的存储区域，那么long数组中就会包含2个元素，总共128位，每一位代表一个区域的占用情况。

相关的代码实现如图15-57所示。

图15-57 PoolSubpage的变量定义

4. 内存回收策略

无论是Chunk还是Page，都通过状态位来标识内存是否可用，不同之处是Chunk通过在二叉树上对节点进行标识实现，Page是通过维护块的使用状态标识来实现。

对于使用者来说，不需要关心内存池的实现细节，也不需要与这些类库打交道，只需要按照API说明正常使用即可。

15.2.6 PooledDirectByteBuf源码分析

PooledDirectByteBuf基于内存池实现，与UnPooledDirectByteBuf的唯一不同就是缓冲区的分配是销毁策略不同，其他功能都是等同的，也就是说，两者唯一的不同就是内存分配策略不同。

1. 创建字节缓冲区实例

由于采用内存池实现，所以新建PooledDirectByteBuffer对象是不能直接new一个实例，而是从内存池中获取，然后设置引用计数器的值，代码如图15-58所示。

图15-58 PooledDirectByteBuffer的创建

直接从内存池Recycler<PooledDirectByteBuffer>中获取PooledDirectByteBuffer对象，然后设置它的引用计数器为1，设置缓冲区最大容量后返回。

2. 复制新的字节缓冲区实例

如果使用者确实需要复制一个新的实例，与原来的PooledDirectByteBuffer独立，则调用它的copy（int index, int length）可以达到上述目标，代码如图15-59所示。

图15-59 PooledDirectByteBuffer的copy方法

首先对索引和长度进行合法性校验，通过之后调用PooledByteBufferAllocator分配一个新的ByteBuffer，由于PooledByteBufferAllocator没有实现directBuffer方法，所以最终会调用到AbstractByteBufferAllocator的directBuffer方法，相关代码如图15-60所示。

图15-60 AbstractByteBufferAllocator的缓冲区分配

newDirectBuffer方法对于不同的子类有不同的实现策略，如果是基于内存池的分配器，它会从内存池中获取可用的ByteBuffer，如果是非池，则直接创建新的ByteBuffer，相关代码实现如图15-61、15-62所示。

图15-61 基于内存池的缓冲区分配

图15-62 非内存池实现直接创建新的缓冲区

通过上述代码对比我们可以看出，基于内存池的实现直接从缓存中获取ByteBuf而不是创建一个新的对象。

3. 子类实现相关的方法

正如UnpooledHeapByteBuf，PooledDirectByteBuf也有子类实现相关的功能，这些方法如图15-63所示。

图15-63 PooledDirectByteBuf实现相关的方法

从上述代码可以看出，当我们操作子类实现相关的方法时，需要对是否支持这些操作进行判断，否则会导致异常。

15.3 ByteBuf相关的辅助类功能介绍

学习完了核心的ByteBuf之后，下面一起继续学习它的一些常用辅助功能类。

15.3.1 ByteBufHolder

ByteBufHolder是ByteBuf的容器，在Netty中，它非常有用，例如HTTP协议的请求消息和应答消息都可以携带消息体，这个消息体在NIO ByteBuffer中就是个ByteBuffer对象，在Netty中就是ByteBuf对象。由于不同的协议消息体可以包含不同的协议字段和功能，因此，需要对ByteBuf进行包装和抽象，不同的子类可以有不同的实现。

为了满足这些定制化的需求，Netty抽象出了ByteBufHolder对象，它包含了一个ByteBuf，另外还提供了一些其他实用的方法，使用者继承ByteBufHolder接口后可以按需封装自己的实现。相关类库的继承关系如图15-64所示。

图15-64 ByteBufHolder继承关系图

15.3.2 ByteBufAllocator

ByteBufAllocator是字节缓冲区分配器，按照Netty的缓冲区实现不同，共有两种不同的分配器：基于内存池的字节缓冲区分配器和普通的字节缓冲区分配器。接口的继承关系如图15-65所示。

图15-65 ByteBufAllocator继承关系图

下面我们给出ByteBufAllocator的主要API功能列表（表15-3）。

表15-3 ByteBufAllocator主要API功能列表

15.3.3 CompositeByteBuf

CompositeByteBuf允许将多个ByteBuf的实例组装到一起，形成一个统一的视图，有点类似于数据库将多个表的字段组装到一起统一用视图展示。

CompositeByteBuf在一些场景下非常有用，例如某个协议POJO对象包含两部分：消息头和消息体，它们都是ByteBuf对象。当需要对消息进行编码的时候需要进行整合，如果使用JDK的默认能力，有以下两种方式。

（1）将某个ByteBuffer复制到另一个ByteBuffer中，或者创建一个新的ByteBuffer，将两者复制到新建的ByteBuffer中；

（2）通过List或数组等容器，将消息头和消息体放到容器中进行统一维护和处理。

上面的做法非常别扭，实际上我们遇到的问题跟数据库中视图解决的问题一致——缓冲区有多个，但是需要统一展示和处理，必须有存放它们的统一容器。为了解决这个问题，Netty提供了CompositeByteBuf。

我们一起简单看下它的实现，如图15-66所示。

图15-66 CompositeByteBuf源码

它定义了一个Component类型的集合，实际上Component就是ByteBuf的包装实现类，它聚合了ByteBuf对象，维护了在集合中的位置偏移量信息等，它的实现如图15-67所示。

图15-67 Component源码

向CompositeByteBuf中新增一个ByteBuf的代码，如图15-68所示。

图15-68 CompositeByteBuf中新增ByteBuf源码

删除增加的ByteBuf源码，如图15-69所示。

图15-69 CompositeByteBuf中删除ByteBuf源码

注意：删除ByteBuf之后，需要更新各个Component的索引偏移量。

15.3.4 ByteBufUtil

ByteBufUtil是一个非常有用的工具类，它提供了一系列静态方法用于操作ByteBuf对象。它的功能列表如图15-70所示。

图15-70 ByteBufUtil工具类

其中最有用的方法就是对字符串的编码和解码，具体如下。

(1) `encodeString(ByteBufAllocator alloc, CharBuffer src, Charset charset)`: 对需要编码的字符串src按照指定的字符集charset进行编码，利用指定的ByteBufAllocator生成一个新的ByteBuf;

(2) `decodeString(ByteBuffer src, Charset charset)`: 使用指定的ByteBuffer和charset进行对ByteBuffer进行解码，获取解码后的字符串。

还有一个非常有用的方法就是`hexDump`，它能够将参数ByteBuf的内容以十六进制字符串的方式打印出来，用于输出日志或者打印码流，方便问题定位，提升系统的可维护性。

hexDump包含了一系列的方法，参数不同，输出的结果也不同，如图15-71所示。

图15-71 hexDump方法

15.4 总结

本章节重点介绍了ByteBuf的API功能和其源码实现，同时介绍了与ByteBuf密切相关的工具类和辅助类，ByteBuf是Netty架构中最重要、最基础的数据结构，熟练地掌握和使用它是学好Netty的基本要求，也是成长为高级Netty开发人员的必经之路。

由于ByteBuf的功能复杂性，它的子类实现非常庞大，在本书中进行穷举是不现实的。读者学习完本章之后，对Netty ByteBuf的设计理念和重要类库的实现原理都有了比较深入的了解。以此为基础，再去学习其他相关联的类库会容易很多。

下个章节，我们继续学习Netty的另两个重要类库：Channel和Unsafe。

第16章 Channel和Unsafe

提起Channel，读者朋友们可能并不陌生——JDK的NIO类库的重要组成部分，就是提供了`java.nio.SocketChannel`和`java.nio.ServerSocketChannel`，用于非阻塞的I/O操作。

类似于NIO的Channel，Netty提供了自己的Channel和其子类实现，用于异步I/O操作和其他相关的操作。

Unsafe是个内部接口，聚合在Channel中协助进行网络读写相关的操作，因为它的设计初衷就是Channel的内部辅助类，不应该被Netty框架的上层使用者调用，所以被命名为Unsafe。这里不能仅从字面理解认为它是不安全的操作，而要从整个架构的设计层面体会它的设计初衷和职责。

本章主要包括：

- Channel 功能说明
- Unsafe功能说明
- Channel的主要实现子类源码分析
- Unsafe的主要实现子类源码分析

16.1 Channel 功能说明

`io.netty.channel.Channel`是Netty网络操作抽象类，它聚合了一组功能，包括但不限于网络的读、写，客户端发起连接、主动关闭连接，链路关闭，获取通信双方的网络地址等。它也包含了Netty框架相关的一些功能，包括获取该Channel的EventLoop，获取缓冲分配器ByteBufAllocator和pipeline等。

下面我们先从Channel的接口分析，讲解它的主要API和功能，然后再一起看下它的子类的相关功能实现，最后再对重要子类和接口进行源码分析。

16.1.1 Channel的工作原理

Channel是Netty抽象出来的网络I/O读写相关的接口，为什么不使用JDK NIO 原生的Channel而要另起炉灶呢，主要原因如下。

（1）JDK的SocketChannel和ServerSocketChannel没有统一的Channel接口供业务开发者使用，对于用户而言，没有统一的操作视图，使用起来并不方便。

（2）JDK的SocketChannel和ServerSocketChannel的主要职责就是网络I/O操作，由于它们是SPI类接口，由具体的虚拟机厂家来提供，所以通过继承SPI功能类来扩展其功能的难度很大；直接实现ServerSocketChannel和SocketChannel抽象类，其工作量和重新开发一个新的Channel功能类是差不多的。

（3）Netty的Channel需要能够跟Netty的整体架构融合在一起，例

如I/O模型、基于ChannelPipeline的定制模型，以及基于元数据描述配置化的TCP参数等，这些JDK的SocketChannel和ServerSocketChannel都没有提供，需要重新封装。

（4）自定义的Channel，功能实现更加灵活。

基于上述4个原因，Netty重新设计了Channel接口，并且给予了很多不同的实现。它的设计原理比较简单，但是功能却比较繁杂，主要的设计理念如下。

（1）在Channel接口层，采用Facade模式进行统一封装，将网络I/O操作、网络I/O相关联的其他操作封装起来，统一对外提供。

（2）Channel接口的定义尽量大而全，为SocketChannel和ServerSocketChannel提供统一的视图，由不同子类实现不同的功能，公共功能在抽象父类中实现，最大程度上实现功能和接口的重用。

（3）具体实现采用聚合而非包含的方式，将相关的功能类聚合在Channel中，由Channel统一负责分配和调度，功能实现更加灵活。

16.1.2 Channel的功能介绍

Channel的功能比较繁杂，我们通过分类的方式对它的主要功能进行介绍。

1. 网络I/O操作

Channel网络I/O相关的方法定义如图16-1所示。

图16-1 Channel的I/O操作API列表

下面我们对这些API的功能进行分类说明，读写相关的API列表。

(1) `Channel read()`: 从当前的Channel中读取数据到第一个inbound缓冲区中，如果数据被成功读取，触发`ChannelHandler.channelRead(ChannelHandlerContext, Object)`事件，读取操作API调用完成之后，紧接着会触发`ChannelHandler.channelReadComplete (Channel HandlerContext)`事件，这样业务的ChannelHandler可以决定是否需要继续读取数据。如果已经有读操作请求被挂起，则后续的读操作会被忽略。

(2) `ChannelFuture write(Object msg)`: 请求将当前的msg通过ChannelPipeline写入到目标Channel中。注意，write操作只是将消息存入到消息发送环形数组中，并没有真正被发送，只有调用flush操作才会被写入到Channel中，发送给对方。

(3) `ChannelFuture write(Object msg, ChannelPromise promise)`: 功能与`write(Object msg)`相同，但是携带了ChannelPromise参数负责设置写入操作的结果。

(4) `ChannelFuture writeAndFlush(Object msg, ChannelPromise promise)`: 与方法(3)功能类似，不同之处在于它会将消息写入到Channel中发送，等价于单独调用write和flush操作的组合。

(5) `ChannelFuture writeAndFlush(Object msg)`: 功能等同于方法(4)，但是没有携带`writeAndFlush(Object msg)`参数。

(6) `Channel flush()`: 将之前写入到发送环形数组中的消息全部写入到目标Channel中，发送给通信对方。

(7) `ChannelFuture close(ChannelPromise promise)`: 主动关闭当前连接, 通过`Channel Promise`设置操作结果并进行结果通知, 无论操作是否成功, 都可以通过`ChannelPromise`获取操作结果。该操作会级联触发`ChannelPipeline`中所有`ChannelHandler`的`ChannelHandler.close(ChannelHandlerContext, ChannelPromise)`事件。

(8) `ChannelFuture disconnect(ChannelPromise promise)`: 请求断开与远程通信对端的连接并使用`ChannelPromise`来获取操作结果的通知消息。该方法会级联触发`ChannelHandler.disconnect(ChannelHandlerContext, ChannelPromise)`事件。

(9) `ChannelFuture connect(SocketAddress remoteAddress)`: 客户端使用指定的服务端地址`remoteAddress`发起连接请求, 如果连接因为应答超时而失败, `ChannelFuture`中的操作结果就是`ConnectTimeoutException`异常, 如果连接被拒绝, 操作结果为`ConnectException`。该方法会级联触发`ChannelHandler.connect(ChannelHandlerContext, SocketAddress, SocketAddress, ChannelPromise)`事件。

(10) `ChannelFuture connect(SocketAddress remoteAddress, SocketAddress localAddress)`: 与方法9) 功能类似, 唯一不同的就是先绑定指定的本地地址`localAddress`, 然后再连接服务端。

(11) `ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise)`: 与方法9) 功能类似, 唯一不同的是携带了`ChannelPromise`参数用于写入操作结果。

(12) `connect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise)`: 与方法(11) 功能类似, 唯一不

同的就是绑定了本地地址。

(13) `ChannelFuture bind(SocketAddress localAddress)`: 绑定指定的本地Socket地址`localAddress`, 该方法会级联触发 `ChannelHandler.bind(ChannelHandlerContext, SocketAddress, ChannelPromise)`事件。

(14) `ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise)`: 与方法13) 功能类似, 多携带了一个 `ChannelPromise`用于写入操作结果。

(15) `ChannelConfig config()`: 获取当前Channel的配置信息, 例如 `CONNECT_TIMEOUT_MILLIS`。

(16) `boolean isOpen()`: 判断当前Channel是否已经打开。

(17) `boolean isRegistered()`: 判断当前Channel是否已经注册到 `EventLoop`上。

(18) `boolean isActive()`: 判断当前Channel是否已经处于激活状态。

(19) `ChannelMetadata metadata()`: 获取当前Channel的元数据描述信息, 包括TCP参数配置等。

(20) `SocketAddress localAddress()`: 获取当前Channel的本地绑定地址。

(21) `SocketAddress remoteAddress()`: 获取当前Channel通信的远程Socket地址。

2. 其他常用的API功能说明

第一个比较重要的方法是`eventLoop()`。Channel需要注册到EventLoop的多路复用器上，用于处理I/O事件，通过`eventLoop()`方法可以获取到Channel注册的EventLoop。EventLoop本质上就是处理网络读写事件的Reactor线程。在Netty中，它不仅仅用来处理网络事件，也可以用来执行定时任务和用户自定义NioTask等任务。

第二个比较常用的方法是`metadata()`方法。熟悉TCP协议的读者可能知道，当创建Socket的时候需要指定TCP参数，例如接收和发送的TCP缓冲区大小，TCP的超时时间，是否重用地址等等。在Netty中，每个Channel对应一个物理连接，每个连接都有自己的TCP参数配置。所以，Channel会聚合一个ChannelMetadata用来对TCP参数提供元数据描述信息，通过`metadata()`方法就可以获取当前Channel的TCP参数配置。

第三个方法是`parent()`。对于服务端Channel而言，它的父Channel为空；对于客户端Channel，它的父Channel就是创建它的ServerSocketChannel。

第四个方法是用户获取Channel标识的`id()`，它返回ChannelId对象，ChannelId是Channel的唯一标识，它的可能生成策略如下。

（1）机器的MAC地址（EUI-48或者EUI-64）等可以代表全局唯一的信息；

（2）当前的进程ID；

（3）当前系统时间的毫秒——`System.currentTimeMillis()`；

- (4) 当前系统时间纳秒数——`System.nanoTime()`;
- (5) 32位的随机整型数;
- (6) 32位自增的序列数。

16.2 Channel源码分析

Channel的实现子类非常多，继承关系复杂，从学习的角度我们抽取最重要的两个——Channel-
`io.netty.channel.socket.nio.NioServerSocketChannel`和
`io.netty.channel.socket.nio.NioSocketChannel`进行重点分析。如果读者对其他的Channel实现细节感兴趣，可以按照本书的指导自行阅读。

16.2.1 Channel的主要继承关系类图

为了便于学习和阅读源码，我们分别看下NioSocketChannel和NioServerSocketChannel的继承关系类图。

服务端NioServerSocketChannel的继承关系类图如图16-2所示。

图16-2 NioServerSocketChannel 继承关系类图

客户端NioSocketChannel的继承关系类图如图16-3所示。

图16-3 NioSocketChannel 继承关系类图

16.2.2 AbstractChannel源码分析

1. 成员变量定义

在分析AbstractChannel源码之前，我们先看下它的成员变量定义，如图16-4所示。

图16-4 AbstractChannel成员变量定义

首先定义了两个静态全局异常。

- `CLOSED_CHANNEL_EXCEPTION`: 链路已经关闭已经异常;
- `NOT_YET_CONNECTED_EXCEPTION`: 物理链路尚未建立异常。

声明完上述两个异常之后, 通过静态块将它们的堆栈设置为空的 `StackTraceElement`。

`estimatorHandle`用于预测下一个报文的大小, 它基于之前数据的采样进行分析预测。

根据之前的Channel原理分析, 我们知道`AbstractChannel`采用聚合的方式封装各种功能, 从成员变量的定义可以看出, 它聚合了以下内容。

- `parent`: 代表父类Channel;
- `id`: 采用默认方式生成的全局唯一ID;
- `unsafe`: `Unsafe`实例;
- `pipeline`: 当前Channel对应的`DefaultChannelPipeline`;
- `eventLoop`: 当前Channel注册的`EventLoop`;

.....

在此不一一枚举。通过变量定义可以看出, `AbstractChannel`聚合了所有Channel使用到的能力对象, 由`AbstractChannel`提供初始化和统一封装, 如果功能和子类强相关, 则定义成抽象方法由子类具体实现, 下面的小节就对它的主要API进行源码分析。

2. 核心API源码分析

首先看下网络读写操作，前面章节介绍到网络I/O操作时讲到它会触发ChannelPipeline中对应的事件方法。Netty基于事件驱动，我们也可以理解为当Channel进行I/O操作时会产生对应的I/O事件，然后驱动事件在ChannelPipeline中传播，由对应的ChannelHandler对事件进行拦截和处理，不关心的事件可以直接忽略。采用事件驱动的方式可以非常轻松地通过事件定义来划分事件拦截切面，方便业务的定制和功能扩展，相比AOP，其性能更高，但是功能却基本等价。

网络I/O操作直接调用DefaultChannelPipeline的相关方法，由DefaultChannelPipeline中对应的ChannelHandler进行具体的逻辑处理，如图16-5所示。

图16-5 AbstractChannel网络I/O操作源码实现

AbstractChannel也提供了一些公共API的具体实现，例如localAddress()和remoteAddress()方法，它的源码实现如图16-6所示。

图16-6 AbstractChannel remoteAddress方法实现

首先从缓存的成员变量中获取，如果第一次调用为空，需要通过unsafe的remoteAddress获取，它是个抽象方法，具体由对应的Channel子类实现。

16.2.3 AbstractNioChannel源码分析

1. 成员变量定义

首先，还是从成员变量定义入手，来了解下它的功能实现，成员变量定义如图16-7所示。

图16-7 AbstractNioChannel 成员变量定义

由于NIO Channel、NioSocketChannel和NioServerSocketChannel需要共用，所以定义了一个java.nio.SocketChannel和java.nio.ServerSocketChannel的公共父类SelectableChannel，用于设置SelectableChannel参数和进行I/O操作。

第二个参数是readInterestOp，它代表了JDK SelectionKey的OP_READ。

随后定义了一个volatile修饰的SelectionKey，该SelectionKey是Channel注册到EventLoop后返回的选择键。由于Channel会面临多个业务线程的并发写操作，当SelectionKey由SelectionKey修改之后，为了能让其他业务线程感知到变化，所以需要使用volatile保证修改的可见性，后面的多线程章节会专门对volatile的使用进行说明。

最后定义了代表连接操作结果的ChannelPromise以及连接超时定时器ScheduledFuture和请求的通信地址信息。

2. 核心API源码分析

我们一起看下在AbstractNioChannel实现的主要API，首先是Channel的注册，如图16-8所示。

图16-8 AbstractNioChannel的注册方法实现

定义一个布尔类型的局部变量selected来标识注册操作是否成功，调用Selectable Channel的register方法，将当前的Channel注册到EventLoop的多路复用器上，Selectable Channel的注册方法定义如图16-9

所示。

图16-9 JDK SelectableChannel 的注册方法定义

注册Channel的时候需要指定监听的网络操作位来表示Channel对哪几类网络事件感兴趣，具体的定义如下。

- `public static final int OP_READ = 1 << 0`: 读操作位；
- `public static final int OP_WRITE = 1 << 2`: 写操作位；
- `public static final int OP_CONNECT = 1 << 3`: 客户端连接服务端操作位；
- `public static final int OP_ACCEPT = 1 << 4`: 服务端接收客户端连接操作位。

`AbstractNioChannel`注册的是0，说明对任何事件都不感兴趣，仅仅完成注册操作。注册的时候可以指定附件，后续Channel接收到网络事件通知时可以从`SelectionKey`中重新获取之前的附件进行处理，此处将`AbstractNioChannel`的实现子类自身当作附件注册。如果注册Channel成功，则返回`selectionKey`，通过`selectionKey`可以从多路复用器中获取Channel对象。

如果当前注册返回的`selectionKey`已经被取消，则抛出`CancelledKeyException`异常，捕获该异常进行处理。如果是第一次处理该异常，调用多路复用器的`selectNow()`方法将已经取消的`selectionKey`从多路复用器中删除掉。操作成功之后，将`selected`置为`true`，说明之前失效的`selectionKey`已经被删除掉。继续发起下一次注册操作，如果成功则退出，如果仍然发生`CancelledKeyException`异常，说明我们无法删除已经被取消的`selectionKey`，按照JDK的API说明，这种意外不应该发

生。如果发生这种问题，则说明可能NIO的相关类库存在不可恢复的BUG，直接抛出CancelledKeyException异常到上层进行统一处理。

下面继续看另一个比较重要的方法：准备处理读操作之前需要设置网络操作位为读，代码如图16-10所示。

图16-10 修改网络操作位为读

先判断下Channel是否关闭，如果处于关闭中，则直接返回。获取当前的SelectionKey进行判断，如果可用，说明Channel当前状态正常，则可以进行正常的操作位修改。将SelectionKey当前的操作位与读操作位进行按位与操作，如果等于0，说明目前并没有设置读操作位，通过interestOps | readInterestOp设置读操作位，最后调用selectionKey的interestOps方法重新设置通道的网络操作位，这样就可以监听网络的读事件了。

实际上，对于读操作位的判断和修改与JDK NIO SelectionKey的相关方法实现是等价的，如图16-11所示。

图16-11 判断当前SelectionKey是否可读

16.2.4 AbstractNioByteChannel源码分析

由于成员变量只有一个Runnable类型的flushTask来负责继续写半包消息，所以对成员变量不再单独进行介绍。

最主要的方法就是doWrite（ChannelOutboundBuffer in），下面一起看看它的实现，由于该方法过长，所以我们按照其逻辑进行拆分介绍。如图16-12所示。

图16-12 doWrite (ChannelOutboundBuffer in) 源码片段1

从发送消息环形数组ChannelOutboundBuffer弹出一条消息，判断该消息是否为空，如果为空，说明消息发送数组中所有待发送的消息都已经发送完成，清除半包标识，然后退出循环。清除半包标识的clearOpWrite方法实现如图16-13所示。

图16-13 清除写半包标识

从当前SelectionKey中获取网络操作位，然后与SelectionKey.OP_WRITE做按位与，如果不等于0，说明当前的SelectionKey是isWritable的，需要清除写操作位。清除方法很简单，就是SelectionKey.OP_WRITE取非之后与原操作位做按位与操作，清除SelectionKey的写操作位。

继续看源码，如果需要发送的消息不为空，则继续处理。如图16-14所示。

图16-14 doWrite (ChannelOutboundBuffer in) 源码片段2

首先判断需要发送的消息是否是ByteBuf类型，如果是，则进行强制类型转换，将其转换成ByteBuf类型，判断当前消息的可读字节数是否为0，如果为0，说明该消息不可读，需要丢弃。从环形发送数组中删除该消息，继续循环处理其他的消息。

声明消息发送相关的成员变量，包括：写半包标识、消息是否全部发送标识、发送的总消息字节数。

这些局部变量创建完成之后，对循环发送次数进行判断，如果为-1，则从Channel配置对象中获取循环发送次数。循环发送次数是指当

一次发送没有完成时（写半包），继续循环发送的次数。设置写半包最大循环次数的原因是当循环发送的时候，I/O线程会一直尝试进行写操作，此时I/O线程无法处理其他的I/O操作，例如读新的消息或者执行定时任务和NioTask等，如果网络I/O阻塞或者对方接收消息太慢，可能会导致线程假死。

继续看循环发送的代码如图16-15所示。

图16-15 doWrite（ChannelOutboundBuffer in）源码片段3

调用doWriteBytes进行消息发送，不同的Channel子类有不同的实现，因此它是抽象方法。如果本次发送的字节数为0，说明发送TCP缓冲区已满，发生了ZERO_WINDOW。此时再次发送仍然可能出现写0字节，空循环会占用CPU的资源，导致I/O线程无法处理其他I/O操作，所以将写半包标识setOpWrite设置为true，退出循环，释放I/O线程。

如果发送的字节数大于0，则对发送总数进行计数。判断当前消息是否已经发送成功（缓冲区没有可读字节），如果发送成功则设置done为true，退出当前循环。

消息发送操作完成之后调用ChannelOutboundBuffer更新发送进度信息，然后对发送结果进行判断。如果发送成功，则将已经发送的消息从发送数组中删除；否则调用incompleteWrite方法，设置写半包标识，启动刷新线程继续发送之前没有发送完全的半包消息（写半包）。如图16-16所示。

图16-16 doWrite（ChannelOutboundBuffer in）源码片段4

处理半包发送任务的方法incompleteWrite的实现如图16-17所示。

图16-17 启动写半包任务线程用于后续继续发送半包消息

首先判断是否需要设置写半包标识，如果需要则调用setOpWrite设置写半包标识，代码如图16-18所示。

图16-18 设置写半包标识

设置写半包标识就是将SelectionKey设置成可写的，通过原操作位与SelectionKey.OP_WRITE做按位或操作即可实现。

如果SelectionKey的OP_WRITE被设置，多路复用器会不断轮询对应的Channel用于处理没有发送完成的半包消息，直到清除SelectionKey的OP_WRITE操作位。因此，设置了OP_WRITE操作位后，就不需要启动独立的Runnable来负责发送半包消息了。

如果没有设置OP_WRITE操作位，需要启动独立的Runnable，将其加入到EventLoop中执行，由Runnable负责半包消息的发送，它的实现很简单，就是调用flush()方法来发送缓冲数组中的消息。

消息发送的另一个分支是文件传输，由于它的实现原理与ByteBuf类似，限于篇幅，在此不再详细说明，感兴趣的读者可以自己独立完成分析。

16.2.5 AbstractNioMessageChannel源码分析

由于AbstractNioMessageChannel没有自己的成员变量，所以我们直接对其方法进行说明。

它的主要实现方法只有一个：doWrite(ChannelOutboundBuffer in)，下面首先看下它的源码，如图16-19所示。

图16-19 AbstractNioMessageChannel消息发送

在循环体内对消息进行发送，从ChannelOutboundBuffer中弹出一条消息进行处理，如果消息为空，说明发送缓冲区为空，所有消息都被发送完成。清除写半包标识，退出循环。

与AbstractNioByteChannel的循环发送类似，利用writeSpinCount对单条消息进行发送，调用doWriteMessage(Object msg, ChannelOutboundBuffer in)判断消息是否发送成功，如果成功，则将发送标识done设置为true，退出循环；否则继续执行循环，直到执行writeSpinCount次。

发送操作完成之后，判断发送结果，如果当前的消息被完全发送出去，则将该消息从缓冲数组中删除；否则设置半包标识，注册SelectionKey.OP_WRITE到多路复用器上，由多路复用器轮询对应的Channel重新发送尚未发送完全的半包消息。

通过代码分析我们发现，AbstractNioMessageChannel和AbstractNioByteChannel的消息发送实现比较相似，不同之处在于：一个发送的是ByteBuf或者FileRegion，它们可以直接被发送；另一个发送的则是POJO对象。

16.2.6 AbstractNioMessageServerChannel源码分析

AbstractNioMessageServerChannel的实现非常简单，它定义了一个EventLoopGroup类型的childGroup，用于给新接入的客户端NioSocketChannel分配EventLoop，它的源码实现如图16-20所示。

图16-20 AbstractNioMessageServerChannel源码

每当服务端接入一个新的客户端连接`NioSocketChannel`时，都会调用`childEventLoopGroup`方法获取`EventLoopGroup`线程组，用于给`NioSocketChannel`分配Reactor线程`EventLoop`，相关分配代码如图16-21所示。

图16-21 通过`childEventLoopGroup`方法进行I/O线程分配

16.2.7 `NioServerSocketChannel`源码分析

`NioServerSocketChannel`的实现比较简单，下面我们重点分析主要API的实现，首先看它的成员变量定义和静态方法，如图16-22所示。

图16-22 `NioServerSocketChannel`成员变量定义

首先创建了静态的`ChannelMetadata`成员变量，然后定义了`ServerSocketChannelConfig`用于配置`ServerSocketChannel`的TCP参数。静态的`newSocket`方法用于通过`ServerSocketChannel`的`open`打开新的`ServerSocketChannel`通道。

接着我们再看下`ServerSocketChannel`相关的接口实现：`isActive`、`remoteAddress`、`javaChannel`和`doBind`，它们的源码如图16-23所示。

图16-23 `NioServerSocketChannel`本地实现相关方法

通过`java.net.ServerSocket`的`isBound`方法判断服务端监听端口是否处于绑定状态，它的`remoteAddress`为空。`javaChannel`的实现是`java.nio.ServerSocketChannel`，服务端在进行端口绑定的时候，可以指定`backlog`，也就是允许客户端排队的最大长度。相关API说明如图16-24所示。

图16-24 ServerSocket的绑定方法

下面继续看服务端Channel的doReadMessages(List<Object> buf)的实现，如图16-25所示。

图16-25 NioServerSocketChannel doReadMessages方法

首先通过ServerSocketChannel的accept接收新的客户端连接，如果SocketChannel不为空，则利用当前的NioServerSocketChannel、EventLoop和SocketChannel创建新的NioSocketChannel，并将其加入到List<Object> buf中，最后返回1，表示服务端消息读取成功。

对于NioServerSocketChannel，它的读取操作就是接收客户端的连接，创建NioSocketChannel对象。

最后看下与服务端Channel无关的接口定义，由于这些方法是客户端Channel相关的，因此，对于服务端Channel无须实现，如果这些方法被误调，则返回UnsupportedOperationException异常，这些方法的源码如图16-26所示。

图16-26 NioServerSocketChannel 不支持的方法列表

16.2.8 NioSocketChannel源码分析

1. 连接操作

我们重点分析与客户端连接相关的API实现，首先看连接方法的实现，如图16-27所示。

图16-27 NioSocketChannel的连接方法

判断本地Socket地址是否为空，如果不为空则调用
`java.nio.channels.SocketChannel. socket().bind()`方法绑定本地地址。如果
绑定成功，则继续调用`java.nio.channels.`
`SocketChannel.connect(SocketAddress remote)`发起TCP连接。对连接结果
进行判断，连接结果有以下三种可能。

(1) 连接成功，返回true;

(2) 暂时没有连接上，服务端没有返回ACK应答，连接结果不确定，返回false;

(3) 连接失败，直接抛出I/O异常。

如果是结果(2)，需要将NioSocketChannel中的selectionKey设置为OP_CONNECT，监听连接网络操作位。如果抛出了I/O异常，说明客户端的TCP握手请求直接被RESET或者被拒绝，此时需要关闭客户端连接，代码如图16-28所示。

图16-28 连接失败，关闭客户端

2. 写半包

分析完连接操作之后，继续分析写操作，由于它的实现比较复杂，所以仍然需要将其拆分后分段进行分析，代码如图16-29所示。

图16-29 NioSocketChannel的写方法片段1

获取待发送的ByteBuf个数，如果小于等于1，则调用父类AbstractNioByteChannel的doWrite方法，操作完成之后退出。

在批量发送缓冲区的信息之前，先对一系列的局部变量进行赋值，首先，获取需要发送的ByteBuffer数组个数nioBufferCnt，然后，从ChannelOutboundBuffer中获取需要发送的总字节数，从NioSocketChannel中获取NIO的SocketChannel，将是否发送完成标识设置为false，将是否有写半包标识设置为false。如图16-30所示。

图16-30 NioSocketChannel的写方法片段2

继续分析循环发送的代码，代码如图16-31所示。

图16-31 NioSocketChannel的写方法片段3

就像循环读一样，我们需要对一次Selector轮询的写操作次数进行上限控制，因为如果TCP的发送缓冲区满，TCP处于KEEP-ALIVE状态，消息会无法发送出去，如果不对上限进行控制，就会长时间地处于发送状态，Reactor线程无法及时读取其他消息和执行排队的Task。所以，我们必须对循环次数上限做控制。

调用NIO SocketChannel的write方法，它有三个参数：第一个是需要发送的ByteBuffer数组，第二个是数组的偏移量，第三个参数是发送的ByteBuffer个数。返回值是写入SocketChannel的字节个数。

下面对写入的字节进行判断，如果为0，说明TCP发送缓冲区已满，很有可能无法再写进去，因此从循环中跳出，同时将写半包标识设置为true，用于向多路复用器注册写操作位，告诉多路复用器有没发完的半包消息，需要轮询出就绪的SocketChannel继续发送。代码如图16-32所示。

图16-32 NioSocketChannel的写方法片段4

发送操作完成后进行两个计算：需要发送的字节数要减去已经发送的字节数；发送的字节总数+已经发送的字节数。更新完这两个变量后，判断缓冲区中所有的消息是否已经发送完成，如果是，则把发送完成标识设置为true同时退出循环。如果没有发送完成，则继续循环。从循环发送中退出之后，首先对发送完成标识done进行判断，如果发送完成，则循环释放已经发送的消息。环形数组的发送缓冲区释放完成后，取消半包标识，告诉多路复用器消息已经全部发送完成。代码如图16-33所示。

图16-33 NioSocketChannel的写方法片段5

当缓冲区中的消息没有发送完成，甚至某个ByteBuffer只发送了几个字节，出现了所谓的“写半包”时，该怎么办？下面我们继续看看Netty是如何处理“写半包”的。如图16-34所示。

图16-34 NioSocketChannel的写方法片段6

首先，循环遍历发送缓冲区，对消息的发送结果进行判断，下面具体展开进行说明。

（1）从ChannelOutboundBuffer弹出第一条发送的ByteBuf，然后获取该ByteBuf的读索引和可读字节数。

（2）对可读字节数和发送的总字节数进行比较，如果发送的字节数大于可读的字节数，说明当前的ByteBuf已经被完全发送出去，更新ChannelOutboundBuffer的发送进度信息，将已经发送的ByteBuf删除，释放相关资源。最后，发送的字节数要减去第一条发送的字节数，得到后续消息发送的总字节数，然后继续循环判断第二条消息、第三条消息.....

(3) 如果可读的消息大于已经发送的总字节数，说明这条消息没有被完整地发送出去，仅仅发送了部分数据报，也就是出现了所谓的“写半包”问题。此时，需要更新可读的索引为当前索引 + 已经发送的总字节数，然后更新ChannelOutboundBuffer的发送进度信息，退出循环。

(4) 如果可读字节数等于已经发送的总字节数，则说明最后一次发送的消息是个整包消息，没有剩余的半包消息待发送。更新发送进度信息，将最后一条已发送的消息从缓冲区中删除，最后退出循环。

循环发送操作完成之后，更新SocketChannel的操作位为OP_WRITE，由多路复用器在下一次轮询中触发SocketChannel，继续处理没有发送完成的半包消息。

3. 读写操作

NioSocketChannel的读写操作实际上是基于NIO的SocketChannel和Netty的ByteBuf封装而成，下面我们首先分析从SocketChannel中读取数据报，如图16-35所示。

图16-35 NioSocketChannel读取数据报

它有两个参数，说明如下。

- java.nio.channels.SocketChannel: JDK NIO的SocketChannel;
- length: ByteBuf的可写最大字节数。

实际上就是从SocketChannel中读取L个字节到ByteBuf中，L为ByteBuf可写的字节数，下面我们看下ByteBuf writeBytes方法的实现，

如图16-36所示。

图16-36 从SocketChannel读取数据报到ByteBuf中

首先分析setBytes(int index, ScatteringByteChannel in, int length)在UnpooledHeapByteBuf中的实现，如图16-37所示。

图16-37 UnpooledHeapByteBuf的set方法实现

从SocketChannel中读取字节数组到缓冲区java.nio.ByteBuffer中，它的起始position为writeIndex，limit为writeIndex+length，JDK ByteBuffer的相关DOC说明如图16-38所示。

图16-38 java.nio.ByteBuffer的read方法

16.3 Unsafe功能说明

Unsafe接口实际上是Channel接口的辅助接口，它不应该被用户代码直接调用。实际的I/O读写操作都是由Unsafe接口负责完成的。下面我们一起看下它的API定义。

表16-1 Unsafe API功能列表

16.4 Unsafe源码分析

实际的网络I/O操作基本都是由Unsafe功能类负责实现的，下面我们一起看下它的主要功能子类和重要的API实现。

16.4.1 Unsafe继承关系类图

首先看下如图16-39所示Unsafe接口的类继承关系图。

图16-39 Unsafe继承关系图

16.4.2 AbstractUnsafe源码分析

1. register方法

register方法主要用于将当前Unsafe对应的Channel注册到EventLoop的多路复用器上，然后调用DefaultChannelPipeline的fireChannelRegistered方法。如果Channel被激活，则调用DefaultChannelPipeline的fireChannelActive方法。源码如图16-40所示。

首先判断当前所在的线程是否是Channel对应的NioEventLoop线程，如果是同一个线程则不存在多线程并发操作问题，直接调用register0进行注册；如果是由用户线程或者其他线程发起的注册操作，则将注册操作封装成Runnable，放到NioEventLoop任务队列中执行。注意：如果直接执行register0方法，会存在多线程并发操作Channel的问题。

图16-40 AbstractUnsafe的register方法

下面继续看register0方法的实现，代码如图16-41所示。

图16-41 AbstractUnsafe的register0方法

首先调用ensureOpen方法判断当前Channel是否打开，如果没有打开则无法注册，直接返回。校验通过后调用doRegister方法，它由AbstractNioUnsafe对应的AbstractNioChannel实现，代码如图16-42所示。

图16-42 AbstractNioChannel的doRegister方法

该方法在前面的AbstractNioChannel源码分析中已经介绍过，此处不再赘述。如果doRegister方法没有抛出异常，则说明Channel注册成功。将ChannelPromise的结果设置为成功，调用ChannelPipeline的fireChannelRegistered方法，判断当前的Channel是否已经被激活，如果已经被激活，则调用ChannelPipeline的fireChannelActive方法。

如果注册过程中发生了异常，则强制关闭连接，将异常堆栈信息设置到ChannelPromise中。

2. bind方法

bind方法主要用于绑定指定的端口，对于服务端，用于绑定监听端口，可以设置backlog参数；对于客户端，主要用于指定客户端Channel的本地绑定Socket地址。代码实现如图16-43所示。

图16-43 AbstractUnsafe的bind方法实现

调用doBind方法，对于NioSocketChannel和NioServerSocketChannel

有不同的实现，客户端的实现代码如图16-44所示。

图16-44 NioSocketChannel的doBind方法实现

服务端的doBind方法实现如图16-45所示。

图16-45 NioServerSocketChannel的doBind方法实现

如果绑定本地端口发生异常，则将异常设置到ChannelPromise中用于通知ChannelFuture，随后调用closeIfClosed方法来关闭Channel。

3. disconnect方法

disconnect用于客户端或者服务端主动关闭连接，它的代码如图16-46所示。

图16-46 AbstractUnsafe的disconnect方法实现

4. close方法

在链路关闭之前需要首先判断是否处于刷新状态，如果处于刷新状态说明还有消息尚未发送出去，需要等到所有消息发送完成再关闭链路，因此，将关闭操作封装成Runnable稍后再执行。如图16-47所示。

图16-47 AbstractUnsafe的close方法片段1

如果链路没有处于刷新状态，需要从closeFuture中判断关闭操作是否完成，如果已经完成，不需要重复关闭链路，设置ChannelPromise的操作结果为成功并返回。

执行关闭操作，将消息发送缓冲数组设置为空，通知JVM进行内存回收。调用抽象方法doClose关闭链路。源码如图16-48所示。

图16-48 AbstractUnsafe的close方法片段2

如果关闭操作成功，设置ChannelPromise结果为成功。如果操作失败，则设置异常对象到ChannelPromise中。

调用ChannelOutboundBuffer的close方法释放缓冲区的消息，随后构造链路关闭通知Runnable放到NioEventLoop中执行。源码如图16-49所示。

图16-49 AbstractUnsafe的close方法片段3

最后，调用deregister方法，将Channel从多路复用器上取消注册，代码实现如图16-50所示。

图16-50 AbstractUnsafe的close方法片段4

NioEventLoop的cancel方法实际将selectionKey对应的Channel从多路复用器上去注册，NioEventLoop的相关代码如图16-51所示。

图16-51 取消Channel的注册

5. write方法

write方法实际上将消息添加到环形发送数组中，并不是真正的写Channel，它的代码如图16-52所示。

图16-52 写操作

如果Channel没有处于激活状态，说明TCP链路还没有真正建立成功，当前Channel存在以下两种状态。

(1) Channel打开，但是TCP链路尚未建立成功：
NOT_YET_CONNECTED_EXCEPTION;

(2) Channel已经关闭：CLOSED_CHANNEL_EXCEPTION。

对链路状态进行判断，给ChannelPromise设置对应的异常，然后调用ReferenceCountUtil的release方法释放发送的msg对象。

如果链路状态正常，则将需要发送的msg和promise放入发送缓冲区中（环形数组）。

6. flush方法

flush方法负责将发送缓冲区中待发送的消息全部写入到Channel中，并发送给通信对方。它的代码如图16-53所示。

图16-53 刷新操作

首先将发送环形数组的unflushed指针修改为tail，标识本次要发送消息的缓冲区范围。然后调用flush0进行发送，由于flush0代码非常简单，我们重点分析 doWrite方法，代码如图16-54所示。

图16-54 doWrite方法代码片段1

首先计算需要发送的消息个数（unflushed - flush），如果只有 1 个消息需要发送，则调用父类的写操作，我们分析AbstractNioByteChannel的doWrite()方法，代码如图16-55所示。

图16-55 doWrite方法代码片段2

因为只有一条消息需要发送，所以直接从ChannelOutboundBuffer中获取当前需要发送的消息，代码如图16-56所示。

图16-56 doWrite方法代码片段3

首先，获取需要发送的消息，如果消息为ByteBuf且它分配的是JDK的非堆内存，则直接返回。对返回的消息进行判断，如果为空，说明该消息已经发送完成并被回收，然后执行清空OP_WRITE操作位的clearOpWrite方法，代码如图16-57所示。

图16-57 doWrite方法代码片段4

继续向下分析，如果需要发送的ByteBuf已经没有可写的字节了，则说明已经发送完成，将该消息从环形队列中删除，然后继续循环，代码如图16-58所示。

图16-58 doWrite方法代码片段5

下面我们分析下ChannelOutboundBuffer的remove方法，如图16-59所示。

图16-59 doWrite方法代码片段6

首先判断环形队列中是否还有需要发送的消息，如果没有，则直接返回。如果非空，则首先获取Entry，然后对其进行资源释放，同时对需要发送的索引 flushed进行更新。所有操作执行完之后，调用decrementPendingOutboundBytes减去已经发送的字节数，该方法跟incrementPendingOutboundBytes类似，会进行发送低水位的判断和事件

通知，此处不再赘述。

我们接着继续对消息的发送进行分析，代码如图16-60所示。

图16-60 doWrite方法代码片段7

首先将半包标识设置为false，从DefaultSocketChannelConfig中获取循环发送的次数，进行循环发送，对发送方法doWriteBytes展开分析，如图16-61所示。

图16-61 doWrite方法代码片段8

ByteBuf的readBytes()方法的功能是将当前ByteBuf中的可写字节数组写入到指定的Channel中。方法的第一个参数是Channel，此处就是SocketChannel，第二个参数是写入的字节数组长度，它等于ByteBuf的可读字节数，返回值是写入的字节个数。由于我们将SocketChannel设置为异步非阻塞模式，所以写操作不会阻塞。

从写操作中返回，需要对写入的字节数进行判断，如果为0，说明TCP发送缓冲区已满，不能继续再向里面写入消息，因此，将写半包标识设置为true，然后退出循环，执行后续排队的其他任务或者读操作，等待下一次selector的轮询继续触发写操作。

对写入的字节数进行累加，判断当前的ByteBuf中是否还有没有发送的字节，如果没有可发送的字节，则将done设置为true，退出循环。

从循环发送状态退出后，首先根据实际发送的字节数更新发送进度，实际就是发送的字节数和需要发送的字节数的一个比值。执行完进度更新后，判断本轮循环是否将需要发送的消息全部发送完成，如果发送完成则将该消息从循环队列中删除；否则，设置多路复用器的

OP_WRITE操作位，用于通知Reactor线程还有半包消息需要继续发送。

16.4.3 AbstractNioUnsafe源码分析

AbstractNioUnsafe是AbstractUnsafe类的NIO实现，它主要实现了connect、finishConnect等方法，下面我们对重点API实现进行源码分析。

1. connect方法

首先获取当前的连接状态进行缓存，然后发起连接操作，代码如图16-62所示。

图16-62 AbstractNioUnsafe的connect方法代码片段1

需要指出的是，SocketChannel执行connect()操作有三种可能的结果。

（1）连接成功，返回true；

（2）暂时没有连接上，服务端没有返回ACK应答，连接结果不确定，返回false；

（3）连接失败，直接抛出I/O异常。

如果是第（2）种结果，需要将NioSocketChannel中的selectionKey设置为OP_CONNECT，监听连接应答消息。

异步连接返回之后，需要判断连接结果，如果连接成功，则触发ChannelActive事件，代码如图16-63所示。

图16-63 AbstractNioUnsafe的connect方法代码片段2

这里对ChannelActive事件处理不再进行详细说明，它最终会将NioSocketChannel中的 selectionKey 设置为SelectionKey.OP_READ，用于监听网络读操作位。

如果没有立即连接上服务端，则执行如图16-64所示分支。

图16-64 AbstractNioUnsafe的connect方法代码片段3

上面的操作有两个目的。

（1）根据连接超时时间设置定时任务，超时时间到之后触发校验，如果发现连接并没有完成，则关闭连接句柄，释放资源，设置异常堆栈并发起去注册。

（2）设置连接结果监听器，如果接收到连接完成通知则判断连接是否被取消，如果被取消则关闭连接句柄，释放资源，发起取消注册操作。

2. finishConnect 方法

客户端接收到服务端的TCP握手应答消息，通过SocketChannel的 finishConnect方法对连接结果进行判断，代码如图16-65所示。

图16-65 AbstractNioUnsafe的finishConnect方法代码片段1

首先缓存连接状态，当前返回false，然后执行doFinishConnect方法判断连接结果，代码如图16-66所示。

图16-66 AbstractNioUnsafe的finishConnect方法代码片段2

通过 SocketChannel的finishConnect 方法判断连接结果，执行该方法返回三种可能结果。

- 连接成功返回true;
- 连接失败返回 false;
- 发生链路被关闭、链路中断等异常，连接失败。

只要连接失败，就抛出 Error()，由调用方执行句柄关闭等资源释放操作，如果返回成功，则执行fulfillConnectPromise 方法，它负责将 SocketChannel修改为监听读操作位，用来监听网络的读事件，代码如图16-67所示。

图16-67 AbstractNioUnsafe的fulfillConnectPromise方法

最后对连接超时进行判断：如果连接超时时仍然没有接收到服务端的 ACK 应答消息，则由定时任务关闭客户端连接，将SocketChannel从Reactor线程的多路复用器上摘除，释放资源，代码如图16-68所示。

图16-68 AbstractNioUnsafe的finishConnect方法代码片段3

16.4.4 NioByteUnsafe源码分析

我们重点分析它的read方法，源码如图16-69所示。

图16-69 NioByteUnsafe的read方法代码片段1

首先，获取NioSocketChannel的SocketChannelConfig，它主要用于设置客户端连接的TCP参数，接口如图16-70所示。

图16-70 SocketChannelConfig的API列表

继续看allocHandle的初始化。如果是首次调用，从SocketChannelConfig的RecvByteBufferAllocator中创建Handle。下面我们对RecvByteBufferAllocator进行简单地代码分析：RecvByteBufferAllocator默认有两种实现，分别是AdaptiveRecvByteBufferAllocator和FixedRecvByteBufferAllocator。由于FixedRecvByteBufferAllocator的实现比较简单，我们重点分析AdaptiveRecvByteBufferAllocator的实现。如图16-71所示。

图16-71 RecvByteBufferAllocator接口的继承关系

顾名思义，AdaptiveRecvByteBufferAllocator指的是缓冲区大小可以动态调整的ByteBuffer分配器。它的成员变量定义如图16-72所示。

图16-72 RecvByteBufferAllocator的成员变量定义

它分别定义了三个系统默认值：最小缓冲区长度64字节、初始容量1024字节、最大容量65536字节。还定义了两个动态调整容量时的步进参数：扩张的步进索引为4、收缩的步进索引为1。

最后，定义了长度的向量表SIZE_TABLE并初始化它，初始值如图16-73所示。

图16-73 SIZE_TABLE的扩张

向量数组的每个值都对应一个Buffer容量，当容量小于512的时候，由于缓冲区已经比较小，需要降低步进值，容量每次下调的幅度要小些；当大于512时，说明需要解码的消息码流比较大，这时采用调大步进幅度的方式减少动态扩张的频率，所以它采用512的倍数进行扩张。

接下来我们重点分析下AdaptiveRecvByteBufAllocator的方法。

方法1: `getSizeTableIndex(final int size)`, 代码如图16-74所示。

图16-74 获取向量表索引

根据容量Size查找容量向量表对应的索引——这是个典型的二分查找法, 由于它的算法非常经典, 也比较简单, 此处不再赘述。

下面我们分析下它的内部静态类HandleImpl, 首先, 还是看下它的成员变量, 如图16-75所示。

图16-75 HandleImpl的成员变量

它有5个成员变量, 分别是: 对应向量表的最小索引、最大索引、当前索引、下一次预分配的Buffer大小和是否立即执行容量收缩操作。

我们重点分析它的`record(int actualReadBytes)`方法: 当NioSocketChannel执行完读操作后, 会计算获得本次轮询读取的总字节数, 它就是参数actualReadBytes, 执行record方法, 根据实际读取的字节数对ByteBuf进行动态伸缩和扩张, 代码如图16-76所示。

图16-76 ByteBuf动态伸缩

首先, 对当前索引做步进缩减, 然后获取收缩后索引对应的容量, 与实际读取的字节数进行比对, 如果发现小于收缩后的容量, 则重新对当前索引进行赋值, 取收缩后的索引和最小索引中的较大者作为最新的索引。然后, 为下一次缓冲区容量分配赋值——新的索引对应容量向量表中的容量。相反, 如果当前实际读取的字节数大于之前预分配的初始

容量，则说明实际分配的容量不足，需要动态扩张。重新计算索引，选取当前索引+扩张步进和最大索引中的较小作为当前索引值，然后对下次缓冲区的容量值进行重新分配，完成缓冲区容量的动态扩张。

通过上述分析我们得知，`AdaptiveRecvByteBufAllocator`就是根据本次读取的实际字节数对下次接收缓冲区的容量进行动态调整。

使用动态缓冲区分配器的优点如下。

(1) `Netty`作为一个通用的NIO框架，并不对用户的应用场景进行假设，可以使用它做流媒体传输，也可以用它做聊天工具。不同的应用场景，传输的码流大小千差万别，无论初始化分配的是32K还是1M，都会随着应用场景的变化而变得不适应。因此，`Netty`根据上次实际读取的码流大小对下次的接收Buffer缓冲区进行预测和调整，能够最大限度的满足不同行业的应用场景。

(2) 性能更高，容量过大会导致内存占用开销增加，后续的Buffer处理性能会下降；容量过小时需要频繁地内存扩张来接收大的请求消息，同样会导致性能下降。

(3) 更节约内存。假如通常情况下请求消息平均值为1M左右，接收缓冲区大小为1.2M；突然某个客户发送了一个10M的流媒体附件，接收缓冲区扩张为10M以接纳该附件，如果缓冲区不能收缩，每次缓冲区创建都会分配10M的内存，但是后续所有的消息都是1M左右，这样会导致内存的浪费，如果并发客户端过多，可能会发生内存溢出，最终宕机。

看完了`AdaptiveRecvByteBufAllocator`，我们继续分析读操作。

首先通过接收缓冲区分配器的Handler计算获得下次预分配的缓冲区容量byteBufCapacity，如图16-77所示。紧接着根据缓冲区容量进行缓冲区分配，Netty的缓冲区种类很多，此处重点介绍的是消息的读取，因此对缓冲区不展开说明。

图16-77 NioByteUnsafe的read方法代码片段2

接收缓冲区ByteBuf分配完成后，进行消息的异步读取，代码如图16-78所示。

图16-78 NioByteUnsafe的read方法代码片段3

它是个抽象方法，具体实现在NioSocketChannel中，代码如图16-79所示。

图16-79 NioByteUnsafe的read方法代码片段4

其中javaChannel()返回的是SocketChannel，代码如图16-80所示。

图16-80 NioByteUnsafe的read方法代码片段5

byteBuf.writableBytes()返回本次可读的最大长度，我们继续展开看最终是如何从Channel中读取码流的，代码如图16-81所示。

图16-81 NioByteUnsafe的read方法代码片段6

对setBytes方法展开分析如图16-82所示。

图16-82 NioByteUnsafe的read方法代码片段7

由于SocketChannel的read方法参数是Java NIO的ByteBuffer，所以，

需要先将Netty的ByteBuf转换成JDK的ByteBuffer，随后调用ByteBuffer的clear方法对指针进行重置用于新消息的读取，随后将position指针指到初始读index，读取的上限设置为index + 读取的长度。最后调用read方法将SocketChannel中就绪的码流读取到ByteBuffer中，完成消息的读取，返回读取的字节数。

完成消息的异步读取后，需要对本次读取的字节数进行判断，有以下三种可能：

- 1) 返回0，表示没有就绪的消息可读；
- 2) 返回值大于0，读到了消息；
- 3) 返回值-1，表示发生了I/O异常，读取失败。

下面我们继续看Netty的后续处理，首先对读取的字节数进行判断，如果等于或者小于0，表示没有就绪的消息可读或者发生了I/O异常，此时需要释放接收缓冲区；如果读取的字节数小于0，则需要将close状态位置位，用于关闭连接，释放句柄资源。置位完成之后，退出循环。源码如图16-83所示。

图16-83 NioByteUnsafe的read方法代码片段8

完成一次异步读之后，就会触发一次ChannelRead事件，这里要特别提醒大家的是：完成一次读操作，并不意味着读到了一条完整的消息，因为TCP底层存在组包和粘包，所以，一次读操作可能包含多条消息，也可能是一条不完整的消息。因此不要把它跟读取的消息个数等同起来。在没有做任何半包处理的情况下，以ChannelRead的触发次数做计数器来进行性能分析和统计，是完全错误的。当然，如果你使用了半

包解码器或者处理了半包，就能够实现一次 `ChannelRead`对应一条完整的消息。

触发和完成`ChannelRead`事件调用之后，将接收缓冲区释放，代码如图16-84所示。

图16-84 `NioByteUnsafe`的`read`方法代码片段9

因为一次读操作未必能够完成TCP缓冲区的全部读取工作，所以，读操作在循环体中进行，每次读取操作完成之后，会对读取的字节数进行累加，代码如图16-85所示。

图16-85 `NioByteUnsafe`的`read`方法代码片段10

在累加之前，需要对长度上限做保护，如果累计读取的字节数已经发生溢出，则将读取到的字节数设置为整型的最大值，然后退出循环。原因是本次循环已经读取过多的字节，需要退出，否则会影响后面排队的Task任务和写操作的执行。如果没有溢出，则执行累加操作。代码如图16-86所示。

图16-86 `NioByteUnsafe`的`read`方法代码片段11

最后，对本次读取的字节数进行判断，如果小于缓冲区可写的容量，说明TCP缓冲区已经没有就绪的字节可读，读取操作已经完成，需要退出循环。如果仍然有未读的消息，则继续执行读操作。连续的读操作会阻塞排在后面的任务队列中待执行的Task，以及写操作，所以，要对连续读操作做上限控制，默认值为16次，无论TCP缓冲区有多少码流需要读取，只要连续16次没有读完，都需要强制退出，等待下次selector轮询周期再执行。如图16-87所示。

图16-87 NioByteUnsafe的read方法代码片段12

完成多路复用器本轮读操作之后，触发ChannelReadComplete事件，随后调用接收缓冲区容量分配器的Hanlder的记录方法，将本次读取的总字节数传入到record()方法中进行缓冲区的动态分配，为下一次读取选取更加合适的缓冲区容量，代码如图16-88所示。

图16-88 NioByteUnsafe的read方法代码片段13

上面我们提到，如果读到的返回值为-1，表明发生了I/O异常，需要关闭连接，释放资源，代码如图16-89所示。

图16-89 NioByteUnsafe的read方法代码片段14

至此，请求消息的异步读取源码我们已经分析完成。

16.5 总结

本章介绍了Netty最重要的接口之一——Channel的设计原理和功能列表，并对其主要实现子类NioSocketChannel和NioServerSocketChannel的源码进行了分析，涉及到了“半包读”和“半包写”的相关知识。

由于Channel的很多I/O操作都是通过其内部聚合的Unsafe接口及其子类实现的，如果不清楚Unsafe相关子类的代码实现，也就无法真正了解清楚Channel的实现。因此本章节对Unsafe的相关实现也进行了源码分析。

事实上，Channel的实现子类还有很多，包括用于处理UDP的DatagramChannel、用于本地测试的EmbeddedChannel等。限于篇幅，本书无法对这些子类的功能和源码进行一一枚举。感兴趣的读者可以通过阅读API文档、学习demo和源码分析相结合的方式掌握这些类库的使用。

第17章 ChannelPipeline和ChannelHandler

Netty的ChannelPipeline和ChannelHandler机制类似于Servlet和Filter过滤器，这类拦截器实际上是职责链模式的一种变形，主要是为了方便事件的拦截和用户业务逻辑的定制。

Servlet Filter是JEE Web应用程序级的Java代码组件，它能够以声明的方式插入到HTTP请求响应的处理过程中，用于拦截请求和响应，以便能够查看、提取或以某种方式操作正在客户端和服务端之间交换的数据。拦截器封装了业务定制逻辑，能够实现对Web应用程序的预处理和事后处理。

过滤器提供了一种面向对象的模块化机制，用来将公共任务封装到可插入的组件中。这些组件通过Web部署配置文件（web.xml）进行声明，可以方便地添加和删除过滤器，无须改动任何应用程序代码或JSP页面，由Servlet进行动态调用。通过在请求/响应链中使用过滤器，可以对应用程序（而不是以任何方式替代）的Servlet或JSP页面提供的核心处理进行补充，而不破坏Servlet或JSP页面的功能。由于是纯Java实现，所以Servlet过滤器具有跨平台的可重用性，使得它们很容易地被部署到任何符合Servlet规范的JEE环境中。

Netty的Channel过滤器实现原理与Servlet Filter机制一致，它将Channel的数据管道抽象为ChannelPipeline，消息在ChannelPipeline中流动和传递。ChannelPipeline持有I/O事件拦截器ChannelHandler的链表，由ChannelHandler对I/O事件进行拦截和处理，可以方便地通过新增和删

除ChannelHandler来实现不同的业务逻辑定制，不需要对已有的ChannelHandler进行修改，能够实现对修改封闭和对扩展的支持。

下面我们对ChannelPipeline和ChannelHandler，以及与之相关的ChannelHandlerContext进行详细介绍和源码分析。

本章主要包括：

- ChannelPipeline功能说明
- ChannelPipeline源码分析
- ChannelHandler功能说明
- ChannelHandler源码分析

17.1 ChannelPipeline功能说明

ChannelPipeline是ChannelHandler的容器，它负责ChannelHandler的管理和事件拦截与调度。

17.1.1 ChannelPipeline的事件处理

图17-1展示了一个消息被ChannelPipeline的ChannelHandler链拦截和处理的全过程，消息的读取和发送处理全流程描述如下。

图17-1 ChannelPipeline对事件流的拦截和处理流程

(1) 底层的SocketChannel调用read()方法读取ByteBuf，触发ChannelRead事件，由I/O线程NioEventLoop调用ChannelPipeline的fireChannelRead(Object msg)方法，将消息（ByteBuf）传输到ChannelPipeline中；

(2) 消息依次被HeadHandler、ChannelHandler1、ChannelHandler2.....TailHandler拦截和处理，在这个过程中，任何ChannelHandler都可以中断当前的流程，结束消息的传递；

(3) 调用ChannelHandlerContext的write方法发送消息，消息从TailHandler开始，途经ChannelHandlerN.....ChannelHandler1、HeadHandler，最终被添加到消息发送缓冲区中等待刷新和发送，在此过程中也可以中断消息的传递，例如当编码失败时，就需要中断流程，构造异常的Future返回。

Netty中的事件分为inbound事件和outbound事件。inbound事件通常

由I/O线程触发，例如TCP链路建立事件、链路关闭事件、读事件、异常通知事件等，它对应图17-1的左半部分。

触发inbound事件的方法如下。

（1）ChannelHandlerContext.fireChannelRegistered(): Channel注册事件；

（2）ChannelHandlerContext.fireChannelActive(): TCP链路建立成功，Channel激活事件；

（3）ChannelHandlerContext.fireChannelRead(Object): 读事件；

（4）ChannelHandlerContext.fireChannelReadComplete(): 读操作完成通知事件；

（5）ChannelHandlerContext.fireExceptionCaught(Throwable): 异常通知事件；

（6）ChannelHandlerContext.fireUserEventTriggered(Object): 用户自定义事件；

（7）ChannelHandlerContext.fireChannelWritabilityChanged(): Channel的可写状态变化通知事件；

（8）ChannelHandlerContext.fireChannelInactive(): TCP连接关闭，链路不可用通知事件。

Outbound事件通常是由用户主动发起的网络I/O操作，例如用户发起的连接操作、绑定操作、消息发送等操作，它对应图17-1的右半部

分。

触发outbound事件的方法如下：

（1）ChannelHandlerContext.bind(SocketAddress, ChannelPromise)：绑定本地地址事件；

（2）ChannelHandlerContext.connect(SocketAddress, SocketAddress, ChannelPromise)：连接服务端事件；

（3）ChannelHandlerContext.write(Object, ChannelPromise)：发送事件；

（4）ChannelHandlerContext.flush()：刷新事件；

（5）ChannelHandlerContext.read()：读事件；

（6）ChannelHandlerContext.disconnect(ChannelPromise)：断开连接事件；

（7）ChannelHandlerContext.close(ChannelPromise)：关闭当前Channel事件。

17.1.2 自定义拦截器

ChannelPipeline通过ChannelHandler接口来实现事件的拦截和处理，由于ChannelHandler中的事件种类繁多，不同的ChannelHandler可能只需要关心其中的某一个或者几个事件，所以，通常ChannelHandler只需要继承ChannelHandlerAdapter类覆盖自己关心的方法即可。

例如，下面的例子展示了拦截Channel Active事件，打印TCP链路建立成功日志，代码如下。

```
public class MyInboundHandler extends ChannelHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        System.out.println("TCP connected!");
        ctx.fireChannelActive();
    }
}
```

下面的例子展示了如何在链路关闭的时候释放资源，示例代码如下。

```
public class MyOutboundHandler extends ChannelHandlerAdapter
    @Override
    public void close(ChannelHandlerContext ctx, ChannelPr
        System.out.println("TCP closing ..");
        Object.release();
        ctx.close(promise);
    }
}
```

17.1.3 构建pipeline

事实上，用户不需要自己创建pipeline，因为使用ServerBootstrap或

者Bootstrap启动服务端或者客户端时，Netty会为每个Channel连接创建一个独立的pipeline。对于使用者而言，只需要将自定义的拦截器加入到pipeline中即可。相关的代码如下。

```
pipeline = ch.pipeline();  
pipeline.addLast("decoder", new MyProtocolDecoder());  
pipeline.addLast("encoder", new MyProtocolEncoder());
```

对于类似编解码这样的ChannelHandler，它存在先后顺序，例如MessageToMessageDecoder，在它之前往往需要有ByteToMessageDecoder将ByteBuf解码为对象，然后对对象做二次解码得到最终的POJO对象。Pipeline支持指定位置添加或者删除拦截器，相关接口定义如图17-2所示。

图17-2 按顺序添加ChannelHandler

17.1.4 ChannelPipeline的主要特性

ChannelPipeline支持运行态动态的添加或者删除ChannelHandler，在某些场景下这个特性非常实用。例如当业务高峰期需要对系统做拥塞保护时，就可以根据当前的系统时间进行判断，如果处于业务高峰期，则动态地将系统拥塞保护ChannelHandler添加到当前的ChannelPipeline中，当高峰期过去之后，就可以动态删除拥塞保护ChannelHandler了。

ChannelPipeline是线程安全的，这意味着N 个业务线程可以并发地操作ChannelPipeline而不存在多线程并发问题。但是，ChannelHandler却不是线程安全的，这意味着尽管ChannelPipeline是线程安全的，但是用

户仍然需要自己保证ChannelHandler的线程安全。

17.2 ChannelPipeline源码分析

ChannelPipeline的代码相对比较简单，它实际上是一个ChannelHandler的容器，内部维护了一个ChannelHandler的链表和迭代器，可以方便地实现ChannelHandler查找、添加、替换和删除。

17.2.1 ChannelPipeline的类继承关系图

ChannelPipeline的类继承关系比较简单，如图17-3所示。

图17-3 ChannelPipeline类继承关系图

17.2.2 ChannelPipeline对ChannelHandler的管理

ChannelPipeline是ChannelHandler的管理容器，负责ChannelHandler的查询、添加、替换和删除。由于它与Map等容器的实现非常类似，所以我们只简单抽取新增接口进行源码分析，其他方法读者可以自行阅读和分析。在ChannelPipeline中添加ChannelHandler方法如图17-4所示。

图17-4 ChannelPipeline添加ChannelHandler方法

直接调用addBefore（ChannelHandlerInvoker invoker, String baseName, final String name, ChannelHandler handler）方法，代码如图17-5所示。

图17-5 ChannelPipeline的addBefore方法

由于ChannelPipeline支持运行期动态修改，因此存在两种潜在的多线程并发访问场景。

- I/O线程和用户业务线程的并发访问；
- 用户多个线程之间的并发访问。

为了保证ChannelPipeline的线程安全性，需要通过线程安全容器或者锁来保证并发访问的安全，此处Netty直接使用了synchronized关键字，保证同步块内的所有操作的原子性。首先根据baseName获取它对应的DefaultChannelHandlerContext，ChannelPipeline维护了ChannelHandler名和ChannelHandlerContext实例的映射关系，代码如图17-6所示。

图17-6 ChannelPipeline的context方法

对新增的ChannelHandler名进行重复性校验，如果已经有同名的ChannelHandler存在，则不允许覆盖，抛出IllegalArgumentExpection("Duplicate handler name: " + name)异常。校验通过之后，使用新增的ChannelHandler等参数构造一个新的DefaultChannelHandlerContext实例，代码如图17-7所示。

图17-7 构造新的DefaultChannelHandlerContext实例

将新创建的DefaultChannelHandlerContext添加到当前的pipeline中，代码如图17-8所示。

图17-8 添加DefaultChannelHandlerContext到pipeline

首先需要对添加的ChannelHandlerContext做重复性校验，校验代码如图17-9所示。

图17-9 ChannelHandlerContext重复性校验

如果ChannelHandlerContext不是可以在多个ChannelPipeline中共享

的，且已经被添加到ChannelPipeline中，则抛出ChannelPipelineException异常。Handler指针修改如图17-10所示。

图17-10 ChannelHandlerContext位置指针迁移图

加入成功之后，缓存ChannelHandlerContext，发送新增ChannelHandlerContext通知消息。

17.2.3 ChannelPipeline的inbound事件

当发生某个I/O事件的时候，例如链路建立、链路关闭、读取操作完成等，都会产生一个事件，事件在pipeline中得到传播和处理，它是事件处理的总入口。由于网络I/O相关的事件有限，因此Netty对这些事件进行了统一抽象，Netty自身和用户的ChannelHandler会对感兴趣的事件进行拦截和处理。

pipeline中以fireXXX命名的方法都是从IO线程流向用户业务Handler的inbound事件，它们的实现因功能而异，但是处理步骤类似，总结如下。

- (1) 调用HeadHandler对应的fireXXX方法；
- (2) 执行事件相关的逻辑操作。

以fireChannelActive方法为例，调用head.fireChannelActive()之后，判断当前的Channel配置是否自动读取，如果为真则调用Channel的read方法，代码如图17-11所示。

图17-11 fireChannelActive方法

17.2.4 ChannelPipeline的outbound事件

由用户线程或者代码发起的I/O操作被称为outbound事件，事实上inbound和outbound是Netty自身根据事件在pipeline中的流向抽象出来的术语，在其他NIO框架中并没有这个概念。

inbound事件相关联的操作如图17-12所示。

图17-12 inbound事件相关方法

Pipeline本身并不直接进行I/O操作，在前面对Channel和Unsafe的介绍中我们知道最终都是由Unsafe和Channel来实现真正的I/O操作的。Pipeline负责将I/O事件通过TailHandler进行调度和传播，最终调用Unsafe的I/O方法进行I/O操作，相关代码实现如图17-13所示。

图17-13 pipeline的客户端连接操作

它直接调用TailHandler的connect方法，最终会调用到HeadHandler的connect方法，代码如图17-14所示。

图17-14 HeadHandler的connect方法

最终由HeadHandler调用Unsafe的connect方法发起真正的连接，pipeline仅仅负责事件的调度。

17.3 ChannelHandler功能说明

ChannelHandler类似于Servlet的Filter过滤器，负责对I/O事件或者I/O操作进行拦截和处理，它可以选择性地拦截和处理自己感兴趣的事件，也可以透传和终止事件的传递。

基于ChannelHandler接口，用户可以方便地进行业务逻辑定制，例如打印日志、统一封装异常信息、性能统计和消息编解码等。

ChannelHandler支持注解，目前支持的注解有两种。

- Sharable: 多个ChannelPipeline共用同一个ChannelHandler;
- Skip: 被Skip注解的方法不会被调用，直接被忽略。

17.3.1 ChannelHandlerAdapter功能说明

对于大多数的ChannelHandler会选择性地拦截和处理某个或者某些事件，其他的事件会忽略，由下一个ChannelHandler进行拦截和处理。这就会导致一个问题：用户ChannelHandler必须要实现ChannelHandler的所有接口，包括它不关心的那些事件处理接口，这会导致用户代码的冗余和臃肿，代码的可维护性也会变差。

为了解决这个问题，Netty提供了ChannelHandlerAdapter基类，它的所有接口实现都是事件透传，如果用户ChannelHandler关心某个事件，只需要覆盖ChannelHandlerAdapter对应的方法即可，对于不关心的，可以直接继承使用父类的方法，这样子类的代码就会非常简洁和清晰。前面几章样例代码中，我们的ChannelHandler都是直接继承自ChannelHandler Adapter，开发起来非常简单和高效。

ChannelHandlerAdapter相关的代码实现如图17-15所示。

图17-15 ChannelHandlerAdapter源码

从图17-15的源码中我们发现这些透传方法被@Skip注解了，这些方法在执行的过程中会被忽略，直接跳到下一个ChannelHandler中执行对应的方法。

17.3.2 ByteToMessageDecoder功能说明

利用NIO进行网络编程时，往往需要将读取到的字节数组或者字节缓冲区解码为业务可以使用的POJO对象。为了方便业务将ByteBuf解码成业务POJO对象，Netty提供了ByteToMessageDecoder抽象工具解码类。

用户的解码器继承ByteToMessageDecoder，只需要实现void decode（ChannelHandler Context ctx, ByteBuf in, List<Object> out）抽象方法即可完成ByteBuf到POJO对象的解码。

由于ByteToMessageDecoder并没有考虑TCP粘包和组包等场景，读半包需要用户解码器自己负责处理。正因为如此，对于大多数场景不会直接继承ByteToMessageDecoder，而是继承另外一些更高级的解码器来屏蔽半包的处理，下面的小节我们会对它们进行一一介绍。

17.3.3 MessageToMessageDecoder功能说明

MessageToMessageDecoder实际上是Netty的二次解码器，它的职责是将一个对象二次解码为其他对象。

为什么称它为二次解码器呢？我们知道，从SocketChannel读取到的

TCP数据报是ByteBuffer，实际就是字节数组，我们首先需要将ByteBuffer缓冲区中的数据报读取出来，并将其解码为Java对象；然后对Java对象根据某些规则做二次解码，将其解码为另一个POJO对象。因为MessageToMessageDecoder在ByteToMessageDecoder之后，所以称之为二次解码器。

二次解码器在实际的商业项目中非常有用，以HTTP+XML协议栈为例，第一次解码往往是将字节数组解码成HttpRequest对象，然后对HttpRequest消息中的消息体字符串进行二次解码，将XML格式的字符串解码为POJO对象，这就用到了二次解码器。类似这样的场景还有很多，不再一一枚举。

事实上，做一个超级复杂的解码器将多个解码器组合成一个大而全的MessageTo MessageDecoder解码器似乎也能解决多次解码的问题，但是采用这种方式的代码可维护性会非常差。例如，如果我们打算在HTTP+XML协议栈中增加一个打印码流的功能，即首次解码获取HttpRequest对象之后打印XML格式的码流。如果采用多个解码器组合，在中间插入一个打印消息体的Handler即可，不需要修改原有的代码；如果做一个大而全的解码器，就需要在解码的方法中增加打印码流的代码，可扩展性和可维护性都会变差。

用户的解码器只需要实现void decode(ChannelHandlerContext ctx, I msg, List<Object> out)抽象方法即可，由于它是将一个POJO解码为另一个POJO，所以一般不会涉及到半包的处理，相对于ByteToMessageDecoder更加简单些。

17.3.4 LengthFieldBasedFrameDecoder功能说明

在编解码章节我们讲过TCP的粘包导致解码的时候需要考虑如何处理半包的问题，前面介绍了Netty提供的半包解码器LineBasedFrameDecoder和DelimiterBased FrameDecoder，现在我们继续学习第三种最通用的半包解码器——LengthFieldBasedFrameDecoder。

如何区分一个整包消息，通常有如下4种做法。

- 固定长度，例如每120个字节代表一个整包消息，不足的前面补零。解码器在处理这类定长消息的时候比较简单，每次读到指定长度的字节后再进行解码。
- 通过回车换行符区分消息，例如FTP协议。这类区分消息的方式多用于文本协议。
- 通过分隔符区分整包消息。
- 通过指定长度来标识整包消息。

如果消息是通过长度进行区分的，LengthFieldBasedFrameDecoder都可以自动处理粘包和半包问题，只需要传入正确的参数，即可轻松搞定“读半包”问题。

下面我们看看如何通过参数组合的不同来实现不同的“半包”读取策略。第一种常用的方式是消息的第一个字段是长度字段，后面是消息体，消息头中只包含一个长度字段。它的消息结构定义如图17-16所示。

图17-16 解码前的字节缓冲区（14字节）

使用以下参数组合进行解码。

- lengthFieldOffset = 0;

- `lengthFieldLength = 2;`
- `lengthAdjustment = 0;`
- `initialBytesToStrip = 0。`

解码后的字节缓冲区内容如图17-17所示。

图17-17 包含消息长度字段（14字节）

因为通过`ByteBuf.readableBytes()`方法我们可以获取当前消息的长度，所以解码后的字节缓冲区可以不携带长度字段，由于长度字段在起始位置并且长度为2，所以将`initialBytesToStrip`设置为2，参数组合修改为：

- `lengthFieldOffset = 0;`
- `lengthFieldLength = 2;`
- `lengthAdjustment = 0;`
- `initialBytesToStrip = 2。`

解码后的字节缓冲区内容如图17-18所示。

图17-18 仅包含消息体（12字节）

从图17-18的解码结果看，解码后的字节缓冲区丢弃了长度字段，仅仅包含消息体，不过通过`ByteBuf.readableBytes()`方法仍然能够获取到长度字段的值。

在大多数的应用场景中，长度仅用来标识消息体的长度，这类协议通常由消息长度字段+消息体组成，如图17-18所示的例子。但是，对于一些协议，长度还包含了消息头的长度。在这种应用场景中，往往需要使用`lengthAdjustment`进行修正，修正后的参数组合方式如下。由于整个

消息的长度往往都大于消息体的长度，所以，lengthAdjustment为负数，图17-19展示了通过指定lengthAdjustment字段来包含消息头的长度。

- lengthFieldOffset = 0;
- lengthFieldLength = 2;
- lengthAdjustment = -2;
- initialBytesToStrip = 0。

图17-19 包含消息头长度的解码

由于协议种类繁多，并不是所有的协议都将长度字段放在消息头的首位，当标识消息长度的字段位于消息头的中间或者尾部时，需要使用lengthFieldOffset字段进行标识，下面的参数组合给出了如何解决消息长度字段不在首位的问题。

- lengthFieldOffset = 2;
- lengthFieldLength = 3;
- lengthAdjustment = 0;
- initialBytesToStrip = 0。

图17-20 通过定义长度偏移量解决长度字段不在首位的问题

由于消息头1的长度为2，所以长度字段的偏移量为2；消息长度字段Length为3，所以lengthFieldLength值为3。由于长度字段仅仅标识消息体的长度，所以lengthAdjustment和initialBytesToStrip都为0。

最后一种场景是长度字段夹在两个消息头之间或者长度字段位于消息头的中间，前后都有其他消息头字段，在这种场景下如果想忽略长度字段以及其前面的其他消息头字段，则可以通过initialBytesToStrip参数

来跳过要忽略的字节长度，它的组合效果如下。

- `lengthFieldOffset = 1;`
- `lengthFieldLength = 2;`
- `lengthAdjustment = 1;`
- `initialBytesToStrip = 3。`

图17-21 `initialBytesToStrip`参数的使用

首先，由于HDR1的长度为1，所以长度字段的偏移量`lengthFieldOffset`为1；长度字段为2个字节，所以`lengthFieldLength`为2。由于长度字段是消息体的长度，解码后如果携带消息头中的字段，则需要使用`lengthAdjustment`进行调整，此处它的值为1，表示的是HDR2的长度，最后由于解码后的缓冲区要忽略长度字段和HDR1部分，所以`lengthAdjustment`为3。解码后的结果为13个字节，HDR1和Length字段被忽略。

事实上，通过4个参数的不同组合，可以达到不同的解码效果，用户在使用过程中可以根据业务的实际情况进行灵活调整。

由于TCP存在粘包和组包问题，所以通常情况下必须自己处理半包消息。利用`LengthFieldBasedFrameDecoder`解码器可以自动解决半包问题，它通常的用法如下。

```
pipeline.addLast("frameDecoder", new LineBasedFrameDecoder(80  
pipeline.addLast("stringDecoder", new StringDecoder(CharsetUt
```

在pipeline中增加`LineBasedFrameDecoder`解码器，指定正确的参数

组合，它可以将Netty的ByteBuf解码成单个的整包消息，后面的业务解码器拿到的就是个完整的数据报，正常进行解码即可，不再需要额外考虑“读半包”问题，方便了业务消息的解码。

17.3.5 MessageToByteEncoder功能说明

MessageToByteEncoder负责将POJO对象编码成ByteBuf，用户的编码器继承MessageToByteEncoder，实现void encode(ChannelHandlerContext ctx, I msg, ByteBuf out)接口接口，示例代码如下。

```
public class IntegerEncoder extends MessageToByteEncoder<Integer> {
    @Override
    public void encode(ChannelHandlerContext ctx, Integer msg, ByteBuf out)
        throws Exception {
        out.writeInt(msg);
    }
}
```

17.3.6 MessageToMessageEncoder功能说明

将一个POJO对象编码成另一个对象，以HTTP+XML协议为例，它的一种实现方式是：先将POJO对象编码成XML字符串，再将字符串编码为HTTP请求或者应答消息。对于复杂协议，往往需要经历多次编码，为了便于功能扩展，可以通过多个编码器组合来实现相关功能。

用户的解码器继承MessageToMessageDecoder解码器，实现void

encode(ChannelHandlerContext ctx, I msg, List<Object> out)方法即可。注意，它与MessageToByteEncoder的区别是输出是对象列表而不是ByteBuf，示例代码如下。

```
public class IntegerToStringEncoder extends MessageToMessage
{
    @Override
    public void encode(ChannelHandlerContext ctx, Integer
        msg, List<Object> out)
        throws Exception
    {
        out.add(message.toString());
    }
}
```

17.3.7 LengthFieldPrepender功能说明

如果协议中的第一个字段为长度字段，Netty提供了LengthFieldPrepender编码器，它可以计算当前待发送消息的二进制字节长度，将该长度添加到ByteBuf的缓冲区头中，如图17-22所示。

图17-22 LengthFieldPrepender编码器

通过LengthFieldPrepender可以将待发送消息的长度写入到ByteBuf的前2个字节，编码后的消息组成为长度字段+原消息的方式。

通过设置LengthFieldPrepender为true，消息长度将包含长度本身占

用的字节数，打开LengthFieldPrepender后，图17-22示例中的编码结果如图17-23所示。

图17-23 打开LengthFieldPrepender开关后的编码结果

17.4 ChannelHandler源码分析

17.4.1 ChannelHandler的类继承关系图

相对于ByteBuf和Channel，ChannelHandler的类继承关系稍微简单些，但是它的子类非常多。由于ChannelHandler是Netty框架和用户代码的主要扩展和定制点，所以它的子类种类繁多、功能各异，系统ChannelHandler主要分类如下。

- ChannelPipeline的系统ChannelHandler，用于I/O操作和对事件进行预处理，对于用户不可见，这类ChannelHandler主要包括HeadHandler和TailHandler；
- 编解码ChannelHandler，包括ByteToMessageCodec、MessageToMessageDecoder等，这些编解码类本身又包含多种子类，如图17-24所示。

图17-24 编解码ChannelHandler

图17-24 编解码ChannelHandler子类继承关系图

- 其他系统功能性ChannelHandler，包括流量整形Handler、读写超时Handler、日志Handler等。

本章节仅给出讲解到的编解码类，其他不再一一枚举。

17.4.2 ByteToMessageDecoder源码分析

顾名思义，ByteToMessageDecoder解码器用于将ByteBuf解码成POJO对象，下面一起看它的实现。

首先看channelRead方法的源码，如图17-25所示。

图17-25 ByteToMessageDecoder的channelRead方法

首先判断需要解码的msg对象是否是ByteBuf，如果是ByteBuf才需要进行解码，否则直接透传。

通过cumulation是否为空判断解码器是否缓存了没有解码完成的半包消息，如果为空，说明是首次解码或者最近一次已经处理完了半包消息，没有缓存的半包消息需要处理，直接将需要解码的ByteBuf赋值给cumulation；如果cumulation缓存有上次没有解码完成的ByteBuf，则进行复制操作，将需要解码的ByteBuf复制到cumulation中，它的原理如下。

半包解码前：（半包消息1= cumulation.readableBytes()）

半包解码后：（半包消息2= msg.readableBytes()）

在复制之前需要对cumulation的可写缓冲区进行判断，如果不足则需要动态扩展，扩展的代码如图17-26所示。

图17-26 ByteToMessageDecoder的expandCumulation方法

扩展的代码很简单，利用字节缓冲区分配器重新分配一个新的ByteBuf，将老的cumulation复制到新的ByteBuf中，释放cumulation。需要注意的是，此处内存扩展没有采用倍增或者步进的方式，分配的缓冲区恰恰够用，此处的算法可以优化下，以防止连续半包导致的频繁缓冲区扩张和内存复制。

复制操作完成之后释放需要解码的ByteBuf对象，调用callDecode方

法进行解码，代码如图17-27所示。

图17-27 ByteToMessageDecoder的callDecode方法

对ByteBuf进行循环解码，循环的条件是解码缓冲区对象中有可读的字节，调用抽象decode方法，由用户的子类解码器进行解码，方法定义如图17-28所示。

图17-28 ByteToMessageDecoder的decode抽象方法

解码后需要对当前的pipeline状态和解码结果进行判断，代码如图17-29所示。

图17-29 ByteToMessageDecoder的callDecode方法

如果当前的ChannelHandlerContext已经被移除，则不能继续进行解码，直接退出循环；如果输出的out列表长度没变化，说明解码没有成功，需要针对以下不同场景进行判断。

1) 如果用户解码器没有消费ByteBuf，则说明是个半包消息，需要由I/O线程继续读取后续的数据报，在这种场景下要退出循环。

2) 如果用户解码器消费了ByteBuf，说明可以解码可以继续进行。

从图17-29所示代码可以看出，业务解码器需要遵守Netty的某些契约，解码器才能正常工作，否则可能会导致功能错误，最重要的契约就是：如果业务解码器认为当前的字节缓冲区无法完成业务层的解码，需要将readIndex复位，告诉Netty解码条件不满足应当退出解码，继续读取数据报。

3) 如果用户解码器没有消费ByteBuf, 但是却解码出了一个或者多个对象, 这种行为被认为是非法的, 需要抛出DecoderException异常。

4) 最后通过isSingleDecode进行判断, 如果是单条消息解码器, 第一次解码完成之后就退出循环。

17.4.3 MessageToMessageDecoder源码分析

MessageToMessageDecoder负责将一个POJO对象解码成另一个POJO对象, 下面一起看下它的源码实现。

首先看channelRead方法的源码, 如图17-30所示。

图17-30 MessageToMessageDecoder的channelRead方法

先通过RecyclableArrayList创建一个新的可循环利用的RecyclableArrayList, 然后对解码的消息类型进行判断, 通过类型参数校验器看是否是可接收的类型, 如果是则校验通过, 参数类型校验的代码如图17-31所示。

图17-31 MessageToMessageDecoder的参数校验

校验通过之后, 直接调用decode抽象方法, 由具体实现子类进行消息解码, 解码抽象方法定义如图17-32所示。

图17-32 MessageToMessageDecoder的抽象decode方法定义

解码完成之后, 调用ReferenceCountUtil的release方法来释放被解码的msg对象。

如果需要解码的对象不是当前解码器可以接收和处理的类型，则将它加入到RecyclableArrayList中不进行解码。

最后，对RecyclableArrayList进行遍历，循环调用ChannelHandlerContext的fireChannelRead方法，通知后续的ChannelHandler继续进行处理。循环通知完成之后，通过recycle方法释放RecyclableArrayList对象。

17.4.4 LengthFieldBasedFrameDecoder源码分析

本节我们一起来学习最通用和重要的解码器——基于消息长度的半包解码器，首先看它的入口方法，源码如图17-33所示。

图17-33 LengthFieldBasedFrameDecoder的decode方法

调用内部的decode(ChannelHandlerContext ctx, ByteBuf in)方法，如果解码成功，将其加入到输出的List<Object> out列表中。

下面继续看decode(ChannelHandlerContext ctx, ByteBuf in)的实现，如图17-34所示。

图17-34 LengthFieldBasedFrameDecoder的decode方法片段1

判断discardingTooLongFrame标识，看是否需要丢弃当前可读的字节缓冲区，如果为真，则执行丢弃操作，具体如下。

判断需要丢弃的字节长度，由于丢弃的字节数不能大于当前缓冲区可读的字节数，所以需要通过Math.min(bytesToDiscard, in.readableBytes())函数进行选择，取bytesToDiscard和缓冲区可读字节数之中的最小值。计算获取需要丢弃的字节数之后，调用ByteBuf的

skipBytes方法跳过需要忽略的字节长度，然后bytesToDiscard减去已经忽略的字节长度。最后判断是否已经达到需要忽略的字节数，达到的话对discardingTooLongFrame等进行置位，代码如图17-35所示。

图17-35 LengthFieldBasedFrameDecoder的failIfNecessary方法

对当前缓冲区的可读字节数和长度偏移量进行对比，如果小于长度偏移量，则说明当前缓冲区的数据报不够，需要返回空，由I/O线程继续读取后续的数据报。如图17-36所示。

图17-36 LengthFieldBasedFrameDecoder的decode方法片段2

通过读索引和lengthFieldOffset计算获取实际的长度字段索引，然后通过索引值获取消息报文的长度字段，代码如图17-37所示。

图17-37 LengthFieldBasedFrameDecoder的getUnadjustedFrameLength方法

根据长度字段自身的字节长度进行判断，共有以下6种可能的取值。

- 长度所占字节为1，通过ByteBuf的getUnsignedByte方法获取长度值；
- 长度所占字节为2，通过ByteBuf的getUnsignedShort方法获取长度值；
- 长度所占字节为3，通过ByteBuf的getUnsignedMedium方法获取长度值；
- 长度所占字节为4，通过ByteBuf的getUnsignedInt方法获取长度值；
- 长度所占字节为8，通过ByteBuf的getLong方法获取长度值；
- 其他长度不支持，抛出DecoderException异常。

获取长度之后，就需要对长度进行合法性判断，同时根据其他解码参数进行长度调整，代码如图17-38所示。

图17-38 LengthFieldBasedFrameDecoder的decode方法片段3

如果长度小于0，说明报文非法，跳过lengthFieldEndOffset个字节，抛出Corrupted FrameException异常。

根据lengthFieldEndOffset和lengthAdjustment字段进行长度修正，如果修正后的报文长度小于lengthFieldEndOffset，则说明是非法数据报，需要抛出CorruptedFrameException异常。

如果修正后的报文长度大于ByteBuf的最大容量，说明接收到的消息长度大于系统允许的最大长度上限，需要设置discardingTooLongFrame，计算需要丢弃的字节数，根据情况选择是否需要抛出解码异常。

丢弃的策略如下：frameLength减去ByteBuf的可读字节数就是需要丢弃的字节长度，如果需要丢弃的字节数discard小于缓冲区可读的字节数，则直接丢弃整包消息。如果需要丢弃的字节数大于当前的可读字节数，说明即便将当前所有可读的字节数全部丢弃，也无法完成任务，则设置discardingTooLongFrame标识为true，下次解码的时候继续丢弃。丢弃操作完成之后，调用failIfNecessary方法根据实际情况抛出异常。如图17-39所示。

图17-39 LengthFieldBasedFrameDecoder的decode方法片段4

如果当前的可读字节数小于frameLength，说明是个半包消息，需要返回空，由I/O线程继续读取后续的数据报，等待下次解码。

对需要忽略的消息头字段进行判断，如果大于消息长度 `frameLength`，说明码流非法，需要忽略当前的数据报，抛出 `CorruptedFrameException` 异常。通过 `ByteBuf` 的 `skipBytes` 方法忽略消息头中不需要的字段，得到整包 `ByteBuf`。

通过 `extractFrame` 方法获取解码后的整包消息缓冲区，代码如图17-40所示。

图17-40 `LengthFieldBasedFrameDecoder`的`extractFrame`方法

根据消息的实际长度分配一个新的 `ByteBuf` 对象，将需要解码的 `ByteBuf` 可写缓冲区复制到新创建的 `ByteBuf` 中并返回，返回之后更新原解码缓冲区 `ByteBuf` 为原读索引+消息报文的实际长度（`actualFrameLength`）。

至此，基于长度的半包解码器介绍完毕，对于使用者而言，实际不需要对 `LengthFieldBasedFrameDecoder` 进行定制。只需要了解每个参数的用法，再结合用户的业务场景进行参数设置，即可实现半包消息的自动解码，后面的业务解码器得到的是个完整的整包消息，不用再额外考虑如何处理半包。这极大地降低了开发难度，提升了开发效率。

17.4.5 `MessageToByteEncoder`源码分析

`MessageToByteEncoder` 负责将用户的 POJO 对象编码成 `ByteBuf`，以便通过网络进行传输。下面一起看它的源码实现，如图17-41所示。

图17-41 `MessageToByteEncoder`的`write`方法片段1

首先判断当前编码器是否支持需要发送的消息，如果不支持则直接

透传；如果支持则判断缓冲区的类型，对于直接内存分配ioBuffer（堆外内存），对于堆内存通过heapBuffer方法分配。

编码使用的缓冲区分配完成之后，调用encode抽象方法进行编码，方法定义如图17-42所示。

图17-42 MessageToByteEncoder的抽象encode方法

编码完成之后，调用ReferenceCountUtil的release方法释放编码对象msg。对编码后的ByteBuf进行以下判断。

- 如果缓冲区包含可发送的字节，则调用ChannelHandlerContext的write方法发送ByteBuf；
- 如果缓冲区没有包含可写的字节，则需要释放编码后的ByteBuf，写入一个空的ByteBuf到ChannelHandlerContext中。

发送操作完成之后，在方法退出之前释放编码缓冲区ByteBuf对象。

17.4.6 MessageToMessageEncoder源码分析

MessageToMessageEncoder负责将一个POJO对象编码成另一个POJO对象，例如将XML Document对象编码成XML格式的字符串。下面一起看它的源码实现，如图17-43所示。

图17-43 MessageToMessageEncoder的抽象write方法

与之前的编码器类似，创建RecyclableArrayList对象，判断当前需要编码的对象是否是编码器可处理的类型，如果不是，则忽略，执行下一个ChannelHandler的write方法。

具体的编码方法实现由用户子类编码器负责完成，如果编码后的RecyclableArrayList为空，说明编码没有成功，释放RecyclableArrayList引用。

如果编码成功，则通过遍历RecyclableArrayList，循环发送编码后的POJO对象，代码如图17-44所示。

图17-44 循环发送编码后的POJO对象

17.4.7 LengthFieldPrepender源码分析

LengthFieldPrepender负责在待发送的ByteBuf消息头中增加一个长度字段来标识消息的长度，它简化了用户的编码器开发，使用户不需要额外去设置这个长度字段。下面我们来看下它的实现，如图17-45所示。

图17-45 LengthFieldPrepender的encode方法片段1

首先对长度字段进行设置，如果需要包含消息长度自身，则在原来长度的基础之上再加上lengthFieldLength的长度。

如果调整后的消息长度小于0，则抛出参数非法异常。对消息长度自身所占的字节数进行判断，以便采用正确的方法将长度字段写入到ByteBuf中，共有以下6种可能。

- 长度字段所占字节为1：如果使用1个Byte字节代表消息长度，则最大长度需要小于256个字节。对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的ByteBuf并通过writeByte将长度值写入到ByteBuf中。

- 长度字段所占字节为2：如果使用2个Byte字节代表消息长度，则最大长度需要小于65536个字节，对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的ByteBuf并通过writeShort将长度值写入到ByteBuf中。
- 长度字段所占字节为3：如果使用3个Byte字节代表消息长度，则最大长度需要小于16777216个字节，对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的ByteBuf并通过writeMedium将长度值写入到ByteBuf中。
- 长度字段所占字节为4：创建新的ByteBuf，并通过writeInt将长度值写入到ByteBuf中。
- 长度字段所占字节为8：创建新的ByteBuf，并通过writeLong将长度值写入到ByteBuf中。
- 其他长度值：直接抛出Error。

最后将原需要发送的ByteBuf复制到List<Object> out中，完成编码。

17.5 总结

本章介绍了ChannelPipeline和ChannelHandler的功能及原理，并给出了使用建议，指出了需要注意的细节。

最后，对ChannelPipeline和ChannelHandler的主要功能子类进行了源码分析。通过学习源码，相信读者不仅仅能学到Netty的一些高级用法，而且能够举一反三，通过按需扩展和功能定制来更好的满足业务的差异化需求。

第18章 EventLoop和EventLoopGroup

从本章开始我们将学习Netty的线程模型。Netty框架的主要线程就是I/O线程，线程模型设计的好坏，决定了系统的吞吐量、并发性和安全性等架构质量属性。

Netty的线程模型被精心地设计，既提升了框架的并发性能，又能在很大程度避免锁，局部实现了无锁化设计。从本章开始，我们将介绍Netty的线程模型，同时对它的NIO线程NioEventLoop进行详尽地源码分析，让读者能够学习到更多I/O相关的多线程设计原理和实现。

本章主要包括：

- Netty的线程模型
- NioEventLoop源码分析

18.1 Netty的线程模型

当我们讨论Netty线程模型的时候，一般首先会想到的是经典的Reactor线程模型，尽管不同的NIO框架对于Reactor模式的实现存在差异，但本质上还是遵循了Reactor的基础线程模型。

下面让我们一起回顾经典的Reactor线程模型。

18.1.1 Reactor单线程模型

Reactor单线程模型，是指所有的I/O操作都在同一个NIO线程上面完成。NIO线程的职责如下。

- 作为NIO服务端，接收客户端的TCP连接；
- 作为NIO客户端，向服务端发起TCP连接；
- 读取通信对端的请求或者应答消息；
- 向通信对端发送消息请求或者应答消息。

Reactor单线程模型如图18-1所示。

图18-1 Reactor单线程模型

由于Reactor模式使用的是异步非阻塞I/O，所有的I/O操作都不会导致阻塞，理论上一个线程可以独立处理所有I/O相关的操作。从架构层面看，一个NIO线程确实可以完成其承担的职责。例如，通过Acceptor类接收客户端的TCP连接请求消息，当链路建立成功之后，通过Dispatch将对应的ByteBuffer派发到指定的Handler上，进行消息解码。用户线程消息编码后通过NIO线程将消息发送给客户端。

在一些小容量应用场景下，可以使用单线程模型。但是这对于高负载、大并发的应用场景却不合适，主要原因如下。

- 一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送。
- 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈。
- 可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。

为了解决这些问题，演进出了Reactor多线程模型。下面我们一起学习下Reactor多线程模型。

18.1.2 Reactor多线程模型

Reactor多线程模型与单线程模型最大的区别就是有一组NIO线程来处理I/O操作，它的原理如图18-2所示。

图18-2 Reactor多线程模型

Reactor多线程模型的特点如下。

- 有专门一个NIO线程——Acceptor线程用于监听服务端，接收客户端的TCP连接请求。
- 网络I/O操作——读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它包含一个任务队列和 N 个可用的线程，

由这些NIO线程负责消息的读取、解码、编码和发送。

- 一个NIO线程可以同时处理 N 条链路，但是一个链路只对应一个NIO线程，防止发生并发操作问题。

在绝大多数场景下，Reactor多线程模型可以满足性能需求。但是，在个别特殊场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如并发百万客户端连接，或者服务端需要对客户端握手进行安全认证，但是认证本身非常损耗性能。在这类场景下，单独一个Acceptor线程可能会存在性能不足的问题，为了解决性能问题，产生了第三种Reactor线程模型——主从Reactor多线程模型。

18.1.3 主从Reactor多线程模型

主从Reactor线程模型的特点是：服务端用于接收客户端连接的不再是一个单独的NIO线程，而是一个独立的NIO线程池。Acceptor接收到客户端TCP连接请求并处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到I/O线程池（sub reactor线程池）的某个I/O线程上，由它负责SocketChannel的读写和编解码工作。Acceptor线程池仅仅用于客户端的登录、握手和安全认证，一旦链路建立成功，就将链路注册到后端subReactor线程池的I/O线程上，由I/O线程负责后续的I/O操作。

它的线程模型如图18-3所示。

图18-3 主从Reactor多线程模型

利用主从NIO线程模型，可以解决一个服务端监听线程无法有效处理所有客户端连接的性能不足问题。因此，在Netty的官方demo中，推荐使用该线程模型。

18.1.4 Netty的线程模型

Netty的线程模型并不是一成不变的，它实际取决于用户的启动参数配置。通过设置不同的启动参数，Netty可以同时支持Reactor单线程模型、多线程模型和主从Reactor多线程模型。

下面让我们通过一张原理图（图18-4）来快速了解Netty的线程模型。

图18-4 Netty的线程模型

可以通过如图18-5所示的Netty服务端启动代码来了解它的线程模型。

图18-5 Netty服务端启动

服务端启动的时候，创建了两个NioEventLoopGroup，它们实际是两个独立的Reactor线程池。一个用于接收客户端的TCP连接，另一个用于处理I/O相关的读写操作，或者执行系统Task、定时任务Task等。

Netty用于接收客户端请求的线程池职责如下。

- （1）接收客户端TCP连接，初始化Channel参数；
- （2）将链路状态变更事件通知给ChannelPipeline。

Netty处理I/O操作的Reactor线程池职责如下。

- （1）异步读取通信对端的数据报，发送读事件到ChannelPipeline；
- （2）异步发送消息到通信对端，调用ChannelPipeline的消息发送接

口；

(3) 执行系统调用Task；

(4) 执行定时任务Task，例如链路空闲状态监测定时任务。

通过调整线程池的线程个数、是否共享线程池等方式，Netty的Reactor线程模型可以在单线程、多线程和主从多线程间切换，这种灵活的配置方式可以最大程度地满足不同用户的个性化定制。

为了尽可能地提升性能，Netty在很多地方进行了无锁化的设计，例如在I/O线程内部进行串行操作，避免多线程竞争导致的性能下降问题。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列—多个工作线程的模型性能更优。

它的设计原理如图18-6所示。

图18-6 Netty Reactor线程模型

Netty的NioEventLoop读取到消息之后，直接调用ChannelPipeline的fireChannelRead (Object msg)。只要用户不主动切换线程，一直都是由NioEventLoop调用用户的Handler，期间不进行线程切换。这种串行化处理方式避免了多线程操作导致的锁的竞争，从性能角度看是最优的。

18.1.5 最佳实践

Netty的多线程编程最佳实践如下。

(1) 创建两个NioEventLoopGroup，用于逻辑隔离NIO Acceptor和NIO I/O线程。

(2) 尽量不要在ChannelHandler中启动用户线程（解码后用于将POJO消息派发到后端业务线程的除外）。

(3) 解码要放在NIO线程调用的解码Handler中进行，不要切换到用户线程中完成消息的解码。

(4) 如果业务逻辑操作非常简单，没有复杂的业务逻辑计算，没有可能会导致线程被阻塞的磁盘操作、数据库操作、网路操作等，可以直接在NIO线程上完成业务逻辑编排，不需要切换到用户线程。

(5) 如果业务逻辑处理复杂，不要在NIO线程上完成，建议将解码后的POJO消息封装成Task，派发到业务线程池中由业务线程执行，以保证NIO线程尽快被释放，处理其他的I/O操作。

推荐的线程数量计算公式有以下两种。

- 公式一：线程数量=（线程总时间/瓶颈资源时间）×瓶颈资源的线程并行数；
- 公式二：QPS=1000/线程总时间×线程数。

由于用户场景的不同，对于一些复杂的系统，实际上很难计算出最优线程配置，只能是根据测试数据和用户场景，结合公式给出一个相对合理的范围，然后对范围内的数据进行性能测试，选择相对最优值。

18.2 NioEventLoop源码分析

18.2.1 NioEventLoop设计原理

Netty的NioEventLoop并不是一个纯粹的I/O线程，它除了负责I/O的读写之外，还兼顾处理以下两类任务。

- 系统Task：通过调用NioEventLoop的execute(Runnable task)方法实现，Netty有很多系统Task，创建它们的主要原因是：当I/O线程和用户线程同时操作网络资源时，为了防止并发操作导致的锁竞争，将用户线程的操作封装成Task放入消息队列中，由I/O线程负责执行，这样就实现了局部无锁化。
- 定时任务：通过调用NioEventLoop的schedule(Runnable command, long delay, TimeUnit unit)方法实现。

正是因为NioEventLoop具备多种职责，所以它的实现比较特殊，它并不是个简单的Runnable。我们来看下它的继承关系，如图18-7所示。

图18-7 NioEventLoop继承关系

它实现了EventLoop接口、EventExecutorGroup接口和ScheduledExecutorService接口，正是因为这种设计，导致NioEventLoop和其父类功能实现非常复杂。下面我们就重点分析它的源码实现，理解它的设计原理。

18.2.2 NioEventLoop继承关系类图

从下个小节开始，我们将对NioEventLoop的源码进行分析。通过源

码分析，希望读者能够理解Netty的Reactor线程设计原理，掌握其精髓。NioEventLoop继承关系图如图18-8所示。

图18-8 NioEventLoop继承关系图

18.2.3 NioEventLoop

作为NIO框架的Reactor线程，NioEventLoop需要处理网络I/O读写事件，因此它必须聚合一个多路复用器对象。下面我们看它的Selector定义，如图18-9所示。

图18-9 NioEventLoop的Selector定义

Selector的初始化非常简单，直接调用Selector.open()方法就能创建并打开一个新的Selector。Netty对Selector的selectedKeys进行了优化，用户可以通过io.netty.noKeySet Optimization开关决定是否启用该优化项。默认不打开selectedKeys优化功能。

Selector的初始化代码如图18-10所示。

图18-10 Selector的初始化

如果没有开启selectedKeys优化开关，通过provider.openSelector()创建并打开多路复用器之后就立即返回。

如果开启了优化开关，需要通过反射的方式从Selector实例中获取selectedKeys和publicSelectedKeys，将上述两个成员变量设置为可写，通过反射的方式使用Netty构造的selectedKeys包装类selectedKeySet将原JDK的selectedKeys替换掉。

分析完Selector的初始化，下面重点看下run方法的实现，如图18-11所示。

图18-11 NioEventLoop的run方法

所有的逻辑操作都在for循环体内进行，只有当NioEventLoop接收到退出指令的时候，才退出循环，否则一直执行下去，这也是通用的NIO线程实现方式。

首先需要将wakenUp还原为false，并将之前的wake up状态保存到oldWakenUp变量中。通过hasTasks()方法判断当前的消息队列中是否有消息尚未处理，如果有则调用selectNow()方法立即进行一次select操作，看是否有准备就绪的Channel需要处理。它的代码实现如图18-12所示。

图18-12 NioEventLoop的selectNow()方法

Selector的selectNow()方法会立即触发Selector的选择操作，如果有准备就绪的Channel，则返回就绪Channel的集合，否则返回0。选择完成之后，再次判断用户是否调用了Selector的wakeup方法，如果调用，则执行selector.wakeup()操作。

下面我们返回到run方法，继续分析代码。如果消息队列中没有消息需要处理，则执行select()方法，由Selector多路复用器轮询，看是否有准备就绪的Channel。它的实现如图18-13所示。

图18-13 NioEventLoop的select()方法

取当前系统的纳秒时间，调用delayNanos()方法计算获得NioEventLoop中定时任务的触发时间。

计算下一个将要触发的定时任务的剩余超时时间，将它转换成毫秒，为超时时间增加0.5毫秒的调整值。对剩余的超时时间进行判断，如果需要立即执行或者已经超时，则调用`selector.selectNow()`进行轮询操作，将`selectCnt`设置为1，并退出当前循环。

将定时任务剩余的超时时间作为参数进行select操作，每完成一次select操作，对select计数器`selectCnt`加1。

Select操作完成之后，需要对结果进行判断，如果存在下列任意一种情况，则退出当前循环。

- 有Channel处于就绪状态，`selectedKeys`不为0，说明有读写事件需要处理；
- `oldWakeup`为true；
- 系统或者用户调用了`wakeup`操作，唤醒当前的多路复用器；
- 消息队列中有新的任务需要处理。

如果本次Selector的轮询结果为空，也没有`wakeup`操作或是新的消息需要处理，则说明是个空轮询，有可能触发了JDK的`epoll bug`，它会导致Selector的空轮询，使I/O线程一直处于100%状态。截止到当前最新的JDK7版本，该bug仍然没有被完全修复。所以Netty需要对该bug进行规避和修正。

Bug-id=6403933的Selector堆栈如图18-14所示。

图18-14 JDK Selector CPU 100% bug

该Bug的修复策略如下。

- (1) 对Selector的select操作周期进行统计；

(2) 每完成一次空的select操作进行一次计数；

(3) 在某个周期（例如100ms）内如果连续发生N次空轮询，说明触发了JDK NIO的epoll()死循环bug。

监测到Selector处于死循环后，需要通过重建Selector的方式让系统恢复正常，代码如图18-15所示。

图18-15 重建Selector

首先通过inEventLoop()方法判断是否是其他线程发起的rebuildSelector，如果由其他线程发起，为了避免多线程并发操作Selector和其他资源，需要将rebuildSelector封装成Task，放到NioEventLoop的消息队列中，由NioEventLoop线程负责调用，这样就避免了多线程并发操作导致的线程安全问题。

调用openSelector方法创建并打开新的Selector，通过循环，将原Selector上注册的SocketChannel从旧的Selector上去注册，重新注册到新的Selector上，并将老的Selector关闭。

相关代码如图18-16所示。

图18-16 将SocketChannel重新注册到新建的Selector上

通过销毁旧的、有问题的多路复用器，使用新建的Selector，就可以解决空轮询Selector导致的I/O线程CPU占用100%的问题。

如果轮询到了处于就绪状态的SocketChannel，则需要处理网络I/O事件，相关代码如图18-17所示。

图18-17 进行I/O操作

由于默认未开启selectedKeys优化功能，所以会进入processSelectedKeysPlain分支执行。下面继续分析processSelectedKeysPlain的代码实现，如图18-18所示。

图18-18 遍历SelectionKey进行网络读写

对SelectionKey进行保护性判断，如果为空则返回。获取SelectionKey的迭代器进行循环操作，通过迭代器获取SelectionKey和SocketChannel的附件对象，将已选择的选择键从迭代器中删除，防止下次被重复选择和处理，代码如图18-19所示。

图18-19 判断SocketChannel的附件类型

对SocketChannel的附件类型进行判读，如果是AbstractNioChannel类型，说明它是NioServerSocketChannel或者NioSocketChannel，需要进行I/O读写相关的操作；如果它是NioTask，则对其进行类型转换，调用processSelectedKey进行处理。由于Netty自身没实现NioTask接口，所以通常情况下系统不会执行该分支，除非用户自行注册该Task到多路复用器。

下面我们继续分析I/O事件的处理，代码如图18-20所示。

图18-20 对选择键的状态进行判断

首先从NioServerSocketChannel或者NioSocketChannel中获取其内部类Unsafe，判断当前选择键是否可用，如果不可用，则调用Unsafe的close方法，释放连接资源。

如果选择键可用，则继续对网络操作位进行判断，代码如图18-21所示。

图18-21 处理读事件

如果是读或者连接操作，则调用Unsafe的read方法。此处Unsafe的实现是个多态，对于NioServerSocketChannel，它的读操作就是接收客户端的TCP连接，相关代码如图18-22所示。

图18-22 NioServerSocketChannel的读操作

对于NioSocketChannel，它的读操作就是从SocketChannel中读取ByteBuffer，相关代码如图18-23所示。

图18-23 NioSocketChannel的读操作

如果网络操作位为写，则说明有半包消息尚未发送完成，需要继续调用flush方法进行发送，相关的代码如图18-24所示。

图18-24 调用flush方法写半包

如果网络操作位为连接状态，则需要对连接结果进行判读，代码如图18-25所示。

图18-25 处理连接事件

需要注意的是，在进行finishConnect判断之前，需要将网络操作位进行修改，注销掉SelectionKey.OP_CONNECT。

处理完I/O事件之后，NioEventLoop需要执行非I/O操作的系统Task和定时任务，代码如图18-26所示。

图18-26 执行非I/O任务

由于NioEventLoop需要同时处理I/O事件和非I/O任务，为了保证两者都能得到足够的CPU时间被执行，Netty提供了I/O比例供用户定制。如果I/O操作多于定时任务和Task，则可以将I/O比例调大，反之则调小，默认值为50%。

Task的执行时间根据本次I/O操作的执行时间计算得来。下面我们具体看runAllTasks方法的实现，如图18-27所示。

图18-27 执行非I/O任务

首先从定时任务消息队列中弹出消息进行处理，如果消息队列为空，则退出循环。根据当前的时间戳进行判断，如果该定时任务已经或者正处于超时状态，则将其加入到执行Task Queue中，同时从延时队列中删除。定时任务如果没有超时，说明本轮循环不需要处理，直接退出即可，代码实现如图18-28所示。

图18-28 从延时队列中移除消息到Task Queue

执行Task Queue中原有的任务和从延时队列中复制的已经超时或者正处于超时状态的定时任务，代码如图18-29所示。

图18-29 执行普通Task和定时任务

由于获取系统纳秒时间是个耗时的操作，每次循环都获取当前系统纳秒时间进行超时判断会降低性能。为了提升性能，每执行60次循环判断一次，如果当前系统时间已经到了分配给非I/O操作的超时时间，则退出循环。这是为了防止由于非I/O任务过多导致I/O操作被长时间阻塞。

最后，判断系统是否进入优雅停机状态，如果处于关闭状态，则需要调用`closeAll`方法，释放资源，并让`NioEventLoop`线程退出循环，结束运行。资源关闭的代码实现如图18-30所示。

图18-30 `NioEventLoop`线程退出，资源释放

遍历获取所有的`Channel`，调用它的`Unsafe.close()`方法关闭所有链路，释放线程池、`ChannelPipeline`和`ChannelHandler`等资源。

18.3 总结

本章详细介绍了Netty的线程模型以及NioEventLoop的实现，通过最佳实践和源码分析，让读者加深对Netty框架线程模型的理解，能够在未来的工作中可以恰到好处地使用它。

对于任何架构，线程模型设计的好坏都直接影响软件的性能和并发处理能力，幸运的是，Netty的线程模型被精心地设计和实现。相信通过对Netty线程模型的学习，广大读者朋友可以举一反三，将Reactor线程模型的精髓应用到日常的工作中。

第19章 **Future和Promise**

本章我们介绍Netty的Future和Promise，读者从名字可以看出，Future用于获取异步操作的结果，而Promise则比较抽象，无法直接猜测出其功能。本章将介绍Netty Future和Promise的功能，分析它们的源码，帮助读者掌握其实现原理。

本章主要包括：

- Future功能
- Future源码分析
- Promise功能
- Promise源码分析

19.1 Future功能

Future最早来源于JDK的`java.util.concurrent.Future`，它用于代表异步操作的结果。相关API如图19-1所示。

图19-1 JDK Future的API列表

可以通过`get`方法获取操作结果，如果操作尚未完成，则会同步阻塞当前调用的线程；如果不允许阻塞太长时间或者无限期阻塞，可以通过带超时时间的`get`方法获取结果；如果到达超时时间操作仍然没有完成，则抛出`TimeoutException`。

通过`isDone()`方法可以判断当前的异步操作是否完成，如果完成，无论成功与否，都返回`true`，否则返回`false`。

通过`cancel`可以尝试取消异步操作，它的结果是未知的，如果操作已经完成，或者发生其他未知的原因拒绝取消，取消操作将会失败。

ChannelFuture功能介绍

由于Netty的Future都是与异步I/O操作相关的，因此，命名为`ChannelFuture`，代表它与`Channel`操作相关。

它的API接口列表如表19-1所示。

表19-1 ChannelFuture接口列表

在Netty中，所有的I/O操作都是异步的，这意味着任何I/O调用都会立即返回，而不是像传统BIO那样同步等待操作完成。异步操作会带来

一个问题：调用者如何获取异步操作的结果？ChannelFuture就是为了解决这个问题而专门设计的。下面我们一起看它的原理。

ChannelFuture有两种状态：uncompleted和completed。当开始一个I/O操作时，一个新的ChannelFuture被创建，此时它处于uncompleted状态——非失败、非成功、非取消，因为I/O操作此时还没有完成。一旦I/O操作完成，ChannelFuture将会被设置成completed，它的结果有如下三种可能。

- 操作成功；
- 操作失败；
- 操作被取消。

ChannelFuture的状态迁移图如图19-2所示。

图19-2 ChannelFuture状态迁移图

ChannelFuture提供了一系列新的API，用于获取操作结果、添加事件监听器、取消I/O操作、同步等待等。

我们重点介绍添加监听器的接口。管理监听器相关的接口定义如图19-3所示。

图19-3 ChannelFuture管理监听器

Netty强烈建议直接通过添加监听器的方式获取I/O操作结果，或者进行后续的相关操作。

ChannelFuture可以同时增加一个或者多个GenericFutureListener，也可以通过remove方法删除GenericFutureListener。

GenericFutureListener的接口定义如图19-4所示。

图19-4 GenericFutureListener接口定义

当I/O操作完成之后，I/O线程会回调ChannelFuture中GenericFutureListener的operationComplete方法，并把ChannelFuture对象当作方法的入参。如果用户需要做上下文相关的操作，需要将上下文信息保存到对应的ChannelFuture中。

推荐通过GenericFutureListener代替ChannelFuture的get等方法的原因是：当我们进行异步I/O操作时，完成的时间是无法预测的，如果不设置超时时间，它会导致调用线程长时间被阻塞，甚至挂死。而设置超时时间，时间又无法精确预测。利用异步通知机制回调GenericFutureListener是最佳的解决方案，它的性能最优。

需要注意的是：不要在ChannelHandler中调用ChannelFuture的await()方法，这会导致死锁。原因是发起I/O操作之后，由I/O线程负责异步通知发起I/O操作的用户线程，如果I/O线程和用户线程是同一个线程，就会导致I/O线程等待自己通知操作完成，这就导致了死锁，这跟经典的两个线程互等待死锁不同，属于自己把自己挂死。

相关代码示例如图19-5所示。

图19-5 ChannelFuture的正反用法示例

异步I/O操作有两类超时：一个是TCP层面的I/O超时，另一个是业务逻辑层面的操作超时。两者没有必然的联系，但是通常情况下业务逻辑超时时间应该大于I/O超时时间，它们两者是包含的关系。

相关代码举例如图19-6所示。

图19-6 I/O超时时间配置

ChannelFuture超时时间配置如图19-7所示。

图19-7 ChannelFuture超时时间配置

需要指出的是：ChannelFuture超时并不代表I/O超时，这意味着ChannelFuture超时后，如果没有关闭连接资源，随后连接依旧可能会成功，这会导致严重的问题。所以通常情况下，必须要考虑究竟是设置I/O超时还是ChannelFuture超时。

19.2 ChannelFuture源码分析

ChannelFuture的接口继承关系如图19-8所示。

图19-8 ChannelFuture接口继承关系图

AbstractFuture

AbstractFuture实现Future接口，它不允许I/O操作被取消。下面我们重点看它的代码实现。

获取异步操作结果的代码如图19-9所示。

图19-9 同步获取I/O操作结果

首先，调用await()方法进行无限期阻塞，当I/O操作完成后会被notify()。程序继续向下执行，检查I/O操作是否发生了异常，如果没有异常，则通过getNow()方法获取结果并返回。否则，将异常堆栈进行包装，抛出ExecutionException。

接着我们看支持超时的获取操作结果方法，如图19-10所示。

图19-10 支持获取超时的方法

支持超时很简单，调用await(long timeout, TimeUnit unit)方法即可。如果超时，则抛出TimeoutException。如果没有超时，则依次判断是否发生了I/O异常等情况，操作与无参数的get方法相同。

其他ChannelFuture的实现子类，由于功能比较简单，读者阅读起来

也没太大难度，所以这里不再花费时间进行详细解读，感兴趣的读者可以独立阅读和分析。`AbstractFuture`的继承关系如图19-11所示。

图19-11 `AbstractFuture`的继承关系图

19.3 Promise功能介绍

Promise是可写的Future，Future自身并没有写操作相关的接口，Netty通过Promise对Future进行扩展，用于设置I/O操作的结果。Future相关的接口定义如图19-12所示。

图19-12 Netty的Future接口定义

Promise相关的写操作接口定义如图19-13所示。

图19-13 Promise写操作相关的接口定义

Netty发起I/O操作的时候，会创建一个新的Promise对象，例如调用ChannelHandler Context的write(Object object)方法时，会创建一个新的ChannelPromise，相关代码如图19-14所示。

图19-14 I/O操作时创建一个新的Promise

当I/O操作发生异常或者完成时，设置Promise的结果，代码如图19-15所示。

图19-15 I/O操作异常时调用tryFailure方法设置结果

19.4 Promise源码分析

19.4.1 Promise继承关系图

由于I/O操作种类非常多，因此对应的Promise子类也非常繁多，它的继承关系如图19-16所示。

图19-16 Promise继承关系图

尽管Promise的子类种类繁多，但是它的功能相对比较清晰，代码也较为简单，因此我们只分析一个它的实现子类的源码，如果读者对其他子类感兴趣，可以自行学习。

19.4.2 DefaultPromise

下面看比较重要的setSuccess方法的实现，如图19-17所示。

图19-17 DefaultPromise的setSuccess方法

首先调用setSuccess0方法并对其操作结果进行判断，如果操作成功，则调用notifyListeners方法通知listener。

setSuccess0方法的实现如图19-18所示。

图19-18 DefaultPromise的setSuccess0私有方法

首先判断当前Promise的操作结果是否已经被设置，如果已经被设置，则不允许重复设置，返回设置失败。

由于可能存在I/O线程和用户线程同时操作Promise，所以设置操作

结果的时候需要加锁保护，防止并发操作。

对操作结果是否被设置进行二次判断（为了提升并发性能的二次判断），如果已经被设置，则返回操作失败。

对操作结果`result`进行判断，如果为空，说明仅仅需要`notify`在等待的业务线程，不包含具体的业务逻辑对象。因此，将`result`设置为系统默认的`SUCCESS`。如果操作结果非空，将结果设置为`result`。

如果有正在等待异步I/O操作完成的用户线程或者其他系统线程，则调用`notifyAll`方法唤醒所有正在等待的线程。注意，`notifyAll`和`wait`方法都必须在同步块内使用。

分析完`setSuccess0`方法，我们继续看`await`方法的实现，如图19-19所示。

图19-19 DefaultPromise的await方法

如果当前的`Promise`已经被设置，则直接返回。如果线程已经被中断，则抛出中断异常。通过同步关键字锁定当前`Promise`对象，使用循环判断对`isDone`结果进行判断，进行循环判断的原因是防止线程被意外唤醒导致的功能异常。如果对循环判断的实现原理感兴趣，读者可以查看《Effective Java中文版第2版》第243页对`wait`和`notify`用法的讲解。

由于在I/O线程中调用`Promise`的`await`或者`sync`方法会导致死锁，所以在循环体中需要对死锁进行保护性校验，防止I/O线程被挂死，最后调用`java.lang.Object.wait()`方法进行无限期等待，直到I/O线程调用`setSuccess`方法、`trySuccess`方法、`setFailure`或者`tryFailure`方法。

19.5 总结

本章重点介绍了Future和Promise，由于Netty中的I/O操作种类繁多，所以Future和Promise的子类也非常繁多。尽管这些子类的功能各异，但本质上都是异步I/O操作结果的通知回调类。Future-Listener机制在JDK中的应用已经非常广泛，所以本章并没有对这些子类的实现做过多的源码分析，希望读者在本章源码分析的基础上自行学习其他相关子类的实现。

无论Future还是Promise，都强烈建议读者通过增加监听器Listener的方式接收异步I/O操作结果的通知，而不是调用wait或者sync阻塞用户线程。

架构和行业应用篇 **Netty**高级特性

第20章 Java多线程编程在Netty中的应用

第21章 Netty架构剖析

第22章 Netty行业应用

第23章 Netty未来展望

第20章 **Java多线程编程在Netty中的应用**

作为异步事件驱动、高性能的NIO框架，Netty代码中大量运用了Java多线程编程技巧。并发编程处理的恰当与否，将直接影响架构的性能。本章通过对Netty源码的分析，结合并发编程的常用技巧，来讲解多线程编程在Netty中的应用。

本章主要包括：

- Java内存模型与多线程编程
- Netty的并发编程剖析

20.1 Java内存模型与多线程编程

20.1.1 硬件的发展和多任务处理

随着硬件，特别是多核处理器的发展和价格的下降，多任务处理已经是所有操作系统必备的一项基本功能。在同一个时刻让计算机做多件事情，不仅是因为处理器的并行计算能力得到了很大提升，还有一个重要的原因是计算机的存储系统、网络通信等I/O性能与CPU的计算能力差距太大，导致程序的很大一部分执行时间被浪费在I/O wait上面，CPU的强大运算能力没有得到充分地利用。

Java提供了很多类库和工具用于降低并发编程的门槛，提升开发效率，一些开源的第三方软件也提供了额外的并发编程类库方便Java开发者，使开发者将重心放在业务逻辑的设计和实现上，而不是处处考虑线程的同步和锁。但是，无论并发类库设计得如何完美，它都无法完全满足用户的需求。对于一个高级Java程序员来说，如果不懂得Java并发编程的原理，只懂得使用一些简单的并发类库和工具，是无法完全驾驭Java多线程这匹野马的。

20.1.2 Java内存模型

JVM规范定义了Java内存模型（Java Memory Model）来屏蔽掉各种操作系统、虚拟机实现厂商和硬件的内存访问差异，以确保Java程序在所有操作系统和平台上能够实现一次编写、到处运行的效果。

Java内存模型的制定既要严谨，保证语义无歧义，还要尽量制定得宽松一些，允许各硬件和虚拟机实现厂商有足够的灵活性来充分利用硬件的特性提升Java的内存访问性能。随着JDK的发展，Java的内存模型

已经逐渐成熟起来。

1. 工作内存和主内存

Java内存模型规定所有的变量都存储在主内存中（JVM内存的一部分），每个线程有自己独立的工作内存，它保存了被该线程使用的变量的主内存复制。线程对这些变量的操作都在自己的工作内存中进行，不能直接操作主内存和其他工作内存中存储的变量或者变量副本。线程间的变量访问需通过主内存来完成，三者的关系如图20-1所示。

图20-1 Java内存访问模型

2. Java内存交互协议

Java内存模型定义了8种操作来完成主内存和工作内存的变量访问，具体如下。

- **lock**: 主内存变量，把一个变量标识为某个线程独占的状态。
- **unlock**: 主内存变量，把一个处于锁定状态变量释放出来，被释放后的变量才可以被其他线程锁定。
- **read**: 主内存变量，把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用。
- **load**: 工作内存变量，把read读取到的主内存中的变量值放入工作内存的变量副本中。
- **use**: 工作内存变量，把工作内存中变量的值传递给Java虚拟机执行引擎，每当虚拟机遇到一个需要使用到变量值的字节码指令时，将会执行该操作。
- **assign**: 工作内存变量，把从执行引擎接收到的变量的值赋值给工

作变量，每当虚拟机遇到一个给变量赋值的字节码时，将会执行该操作。

- **store**: 工作内存变量，把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用。
- **write**: 主内存变量，把store操作从工作内存中得到的变量值放入主内存的变量中。

3. Java的线程

并发可以通过多种方式来实现，例如：单进程—单线程模型，通过在一台服务器上启动多个进程来实现多任务的并行处理。但是在Java语言中，通常是通过单进程—多线程的模型进行多任务的并发处理。因此，我们有必要熟悉一下Java的线程。

大家都知道，线程是比进程更轻量级的调度执行单元，它可以把进程的资源分配和调度执行分开，各个线程可以共享内存、I/O等操作系统资源，但是又能够被操作系统发起的内核线程或者进程执行。各线程可以独立地启动、运行和停止，实现任务的解耦。

主流的操作系统提供了线程实现，目前实现线程的方式主要有三种，分别如下。

(1) 内核线程（KLT）实现，这种线程由内核来完成线程切换，内核通过线程调度器对线程进行调度，并负责将线程任务映射到不同的处理器上。

(2) 用户线程实现（UT），通常情况下，用户线程指的是完全建立在用户空间线程库上的线程，用户线程的创建、启动、运行、销毁和

切换完全在用户态中完成，不需要内核的帮助，因此执行性能更高。

（3）混合实现，将内核线程和用户线程混合在一起使用的方式。

由于虚拟机规范并没有强制规定Java的线程必须使用哪种方式实现，因此，不同的操作系统实现的方式也可能存在差异。对于SUN的JDK，在Windows和Linux操作系统上采用了内核线程的实现方式，在Solaris版本的JDK中，提供了一些专有的虚拟机线程参数，用于设置使用哪种线程模型。

20.2 Netty的并发编程实践

20.2.1 对共享的可变数据进行正确的同步

关键字`synchronized`可以保证在同一时刻，只有一个线程可以执行某一个方法或者代码块。同步的作用不仅仅是互斥，它的另一个作用就是共享可变性，当某个线程修改了可变数据并释放锁后，其他线程可以获取被修改变量的最新值。如果没有正确的同步，这种修改对其他线程是不可见的。

下面我们就通过对Netty源码的分析，看看Netty是如何对并发可变数据进行正确同步的。

以`ServerBootstrap`为例进行分析，首先看它的`option`方法，如图20-2所示。

图20-2 同步关键字的使用

这个方法的作用是设置`ServerBootstrap`的`ServerSocketChannel`的`Socket`属性，它的属性集定义如下。

由于是非线程安全的`LinkedHashMap`，所以当多线程创建、访问和修改`LinkedHashMap`时，必须在外部进行必要的同步。`LinkedHashMap`的API DOC对于线程安全的说明如图20-3所示。

图20-3 LinkedHashMap线程安全API说明

由于`ServerBootstrap`是被外部使用者创建和使用的，我们无法保证它的方法和成员变量不被并发访问，因此，作为成员变量的`options`必须

进行正确地同步。由于考虑到锁的范围需要尽可能的小，我们对传参的 `option` 和 `value` 的合法性判断不需要加锁。因此，代码才对两个判断分支独立加锁，保证锁的范围尽可能的细粒度。

Netty加锁的地方非常多，大家在阅读代码的时候可能会有体会，为什么有的地方要加锁，有的地方有不需要？如果不需要，为什么？当你对锁的使用原理理解以后，对于这些锁的使用时机和技巧理解起来就非常容易了。

20.2.2 正确的使用锁

很多刚接触多线程编程的开发者，虽然意识到了并发访问可变变量需要加锁，但是对于锁的范围、加锁的时机和锁的协同缺乏认识，往往会导致出现一些问题。下面我就结合Netty的代码来讲解下这方面的知识。

打开 `ForkJoinTask`，我们学习一些多线程同步和协作方面的技巧。首先是当条件不满足时阻塞某个任务，直到条件满足后再继续执行，代码如图20-4所示。

图20-4 多线程协作

重点看框线中的代码，首先通过循环检测的方式对状态变量 `status` 进行判断，当它的状态大于等于0时，执行 `wait()`，阻塞当前的调度线程，直到 `status` 小于0，唤醒所有被阻塞的线程，继续执行。这个方法有以下三个多线程的编程技巧需要说明。

(1) `wait` 方法用来使线程等待某个条件，它必须在同步块内部被调用，这个同步块通常会锁定当前对象实例。下面是这个模式的标准使

用方式。

```
synchronized (this)
{
    while(condition)
        Object.wait;
    .....
}
```

(2) 始终使用`wait`循环来调用`wait`方法，永远不要在循环之外调用`wait`方法。这样做的原因是尽管并不满足被唤醒条件，但是由于其他线程调用`notifyAll()`方法会导致被阻塞线程意外唤醒，此时执行条件并不满足，它将破坏被锁保护的约定关系，导致约束失效，引起意想不到的结果。

(3) 唤醒线程，应该使用`notify`还是`notifyAll`，当你不知道究竟该调用哪个方法时，保守的做法是调用`notifyAll`唤醒所有等待的线程。从优化的角度看，如果处于等待的所有线程都在等待同一个条件，而每次只有一个线程可以从这个条件中被唤醒，那么就应该选择调用`notify`。

当多个线程共享同一个变量的时候，每个读或者写数据的操作方法都必须加锁进行同步，如果没有正确的同步，就无法保证一个线程所做的修改被其他线程共享。未能同步共享变量会造成程序的活性失败和安全性失败，这样的失败通常是难以调试和重现的，它们可能间歇性地出问题，可能随着并发的线程个数增加而失败，也可能在不同的虚拟机或者操作系统上存在不同的失败概率。因此，务必要保证锁的正确使用。下面这个案例，就是个典型的错误应用。

```
int size = 0;
public synchronized void increase()
{
    size++;
}
public int current()
{
    Return size;
}
```

20.2.3 volatile的正确使用

长久以来大家对于volatile如何正确使用有很多的争议，即便是一些经验丰富的Java设计师，对于volatile和多线程编程的认识仍然存在误区。其实，volatile的使用非常简单，只要理解了Java的内存模型和多线程编程的基础知识，正确使用volatile是不存在任何问题的。下面我们结合Netty的源码，对volatile的正确使用进行说明。

打开NioEventLoop的代码，我们来看控制I/O操作和其他任务运行比例的ioRatio，它是int类型的变量，定义如下。

我们发现，它被定义为volatile，为什么呢？我们首先对volatile关键字进行说明，然后再结合Netty的代码进行分析。

关键字volatile是Java提供的最轻量级的同步机制，Java内存模型对volatile专门定义了一些特殊的访问规则。下面我们就看它的规则。

当一个变量被volatile修饰后，它将具备以下两种特性。

- 线程可见性：当一个线程修改了被volatile修饰的变量后，无论是否加锁，其他线程都可以立即看到最新的修改，而普通变量却做不到这点。
- 禁止指令重排序优化，普通的变量仅仅保证在该方法的执行过程中所有依赖赋值结果的地方都能获取正确的结果，而不能保证变量赋值操作的顺序与程序代码的执行顺序一致。举个简单的例子说明下指令重排序优化问题，如图20-4所示。

图20-4 指令重排序和优化导致线程无法退出

我们预期程序会在3s后停止，但是实际上它会一直执行下去，原因就是虚拟机对代码进行了指令重排序和优化，优化后的指令如下。

```
if (!stop)
while(true)
.....
```

重排序后的代码是无法发现stop被主线程修改的，因此无法停止运行。要解决这个问题，只要将stop前增加volatile修饰符即可。代码修改如图20-5所示。

图20-5 volatile解决指令重排序和编译优化问题

再次运行，我们发现3s后程序退出，达到了预期效果，使用volatile解决了如下两个问题。

- main线程对stop的修改在workThread线程中可见，也就是说workThread线程立即看到了其他线程对于stop变量的修改。
- 禁止指令重排序，防止因为重排序导致的并发访问逻辑混乱。

一些人认为使用volatile可以代替传统锁，提升并发性能，这个认识是错误的。volatile仅仅解决了可见性的问题，但是它并不能保证互斥性，也就是说多个线程并发修改某个变量时，依旧会产生多线程问题。因此，不能靠volatile来完全替代传统的锁。

根据经验总结，volatile最适合使用的是一个线程写、其他线程读的场合，如果有多个线程并发写操作，仍然需要使用锁或者线程安全的容器或者原子变量来代替。

讲了volatile的原理之后，我们继续对Netty的源码做分析。上面讲到了ioRatio被定义成volatile，下面看看代码为什么要这样定义。参见如图20-6所示代码。

图20-6 volatile在NioEventLoop线程中的应用

通过代码分析我们发现，在NioEventLoop线程中，ioRatio并没有被修改，它是只读操作。既然没有修改，为什么要定义成volatile呢？继续看代码，我们发现NioEventLoop提供了重新设置I/O执行时间比例的公共方法，接口如图20-7所示。

图20-7 修改volatile变量

首先，NioEventLoop线程没有调用该方法，说明调整I/O执行时间比例是外部发起的操作，通常是由业务的线程调用该方法，重新设置该参数。这样就形成了一个线程写、一个线程读，根据前面针对volatile的

应用总结，此时可以使用volatile来代替传统的synchronized关键字提升并发访问的性能。

Netty中大量使用了volatile来修改成员变量，如果理解了volatile的应用场景，读懂Netty volatile的相关代码还是比较容易的。

20.2.4 CAS指令和原子类

互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能的外损耗，因此这种同步被称为阻塞同步，它属于一种悲观的并发策略，我们称之为悲观锁。随着硬件和操作系统指令集的发展和优化，产生了非阻塞同步，被称为乐观锁。简单地说，就是先进行操作，操作完成之后再判断操作是否成功，是否有并发问题，如果有则进行失败补偿，如果没有就算操作成功，这样就从根本上避免了同步锁的弊端。

目前，在Java中应用最广泛的非阻塞同步就是CAS，在IA64、X86指令集中通过cmpxchg指令完成CAS功能，在sparc-TSO中由case指令完成，在ARM和PowerPC架构下，需要使用一对ldrex/strex指令完成。

从JDK1.5以后，可以使用CAS操作，该操作由sun.misc.Unsafe类里的compareAndSwapInt()和compareAndSwapLong()等方法包装提供。通常情况下sun.misc.Unsafe类对于开发者是不可见的，因此，JDK提供了很多CAS包装类简化开发者的使用，如AtomicInteger。

下面，结合Netty的源码，我们对原子类的正确使用进行详细说明。

打开ChannelOutboundBuffer的代码，看看如何对发送的总字节数进行计数和更新操作，先看定义，如图20-8所示。

图20-8 原子类的应用

首先定义了一个`volatile`的变量，它可以保证某个线程对于`totalPendingSize`的修改可以被其他线程立即访问，但是，它无法保证多线程并发修改的安全性。紧接着又定义了一个`AtomicIntegerFieldUpdater`类型的变量`WTOTAL_PENDING_SIZE_UPDATER`，实现`totalPendingSize`的原子更新，也就是保证`totalPendingSize`的多线程修改并发安全性，我们重点看`AtomicIntegerFieldUpdater`的API说明，如图20-9所示。

图20-9 `AtomicIntegerFieldUpdater` Java DOC说明

从API的说明可以看出，它主要用于实现`volatile`修饰的`int`变量的原子更新操作，对于使用者，必须通过类似`compareAndSet`或者`set`或者与这些操作等价的原子操作来保证更新的原子性，否则会导致问题。

继续看代码，当执行`write`操作外发消息的时候，需要对外发的消息字节数进行统计汇总。由于调用`write`操作的既可以是I/O线程，也可以是业务的线程，还可能由业务线程池多个工作线程同时执行发送任务，因此，统计操作是多线程并发的，这也就是为什么要将计数器定义成`volatile`并使用原子更新类进行原子操作。下面看计数的代码，如图20-10所示。

图20-10 通过自旋对计数器进行更新

首先，我们发现计数操作并没有使用锁，而是利用CAS自旋操作，通过`TOTAL_PENDING_SIZE_UPDATER.compareAndSet(this, oldValue, newWriteBufferSize)`来判断本次原子操作是否成功，如果成功则退出循环，代码继续执行；如果失败，说明在本次操作的过程中计数器已经被

其他线程更新成功，需要进入循环，首先对oldValue进行更新，代码如下。

```
oldValue = totalPendingSize;
```

然后重新对更新值进行计算。

```
newWriteBufferSize = oldValue + size;
```

继续循环进行CAS操作，直到成功。它跟AtomicInteger的compareAndSet操作类似。

使用Java自带的Atomic原子类，可以避免同步锁带来的并发访问性能降低的问题，减少犯错的机会，因此，Netty中对于int、long、boolean等成员变量大量使用其原子类，减少了锁的应用，从而降低了频繁使用同步锁带来的性能下降。

20.2.5 线程安全类的应用

在JDK1.5的发行版本中，Java平台新增了java.util.concurrent，这个包中提供了一系列的线程安全集合、容器和线程池，利用这些新的线程安全类可以极大地降低Java多线程编程的难度，提升开发效率。

新的并发编程包中的工具可以分为如下4类。

- 线程池Executor Framework以及定时任务相关的类库，包括Timer等。

- 并发集合，包括List、Queue、Map和Set等。
- 新的同步器，例如读写锁ReadWriteLock等。
- 新的原子包装类，例如AtomicInteger等。

在实际编码过程中，我们建议通过使用线程池、Task（Runnable/Callable）、原子类和线程安全容器来代替传统的同步锁、wait和notify，以提升并发访问的性能、降低多线程编程的难度。

下面，针对新的线程并发包在Netty中的应用进行分析和说明，以期为大家的学习和应用提供指导。

首先看下线程安全容器在Netty中的应用。NioEventLoop是I/O线程，负责网络读写操作，同时也执行一些非I/O的任务。例如事件通知、定时任务执行等，因此，它需要一个任务队列来缓存这些Task。它的任务队列定义如图20-11所示。

图20-11 线程任务队列定义

它是一个ConcurrentLinkedQueue，我们看它的API说明，如图20-12所示。

图20-12 ConcurrentLinkedQueue线程安全文档

DOC文档明确说明这个类是线程安全的，因此，对它进行读写操作不需要加锁。下面我们继续看下队列中增加一个任务，如图20-13所示。

图20-13 ConcurrentLinkedQueue新增Task

读取任务，也不需要加锁，如图20-14所示。

图20-14 ConcurrentLinkedQueue读取Task

JDK的线程安全容器底层采用了CAS、volatile和ReadWriteLock实现，相比于传统重量级的同步锁，采用了更轻量、细粒度的锁，因此，性能会更高。合理地应用这些线程安全容器，不仅能提升多线程并发访问的性能，还能降低开发难度。

下面我们看看线程池在Netty中的应用，打开SingleThreadEventExecutor看它是如何定义和使用线程池的。

首先定义了一个标准的线程池用于执行任务，代码如下。

接着对它赋值并且进行初始化操作，代码如下。

执行任务代码如图20-15所示。

图20-15 SingleThreadEventExecutor任务执行

我们发现，实际上执行任务就是先把任务加入到任务队列中，然后判断线程是否已经启动循环执行，如果不是则需要启动线程。启动线程代码如图20-16所示。

图20-16 SingleThreadEventExecutor启动新的线程

实际上就是执行当前线程的run方法，循环从任务队列中获取Task并执行，我们看它的子类NioEventLoop的run方法就能一目了然，如图20-17所示。

图20-17 按照I/O任务比例执行任务Task

如图20-18中框线内所示，循环从任务队列中获取任务并执行。

图20-18 循环从任务队列中获取任务Task并执行

Netty对JDK的线程池进行了封装和改造，但是，本质上仍然是利用了线程池和线程安全队列简化了多线程编程。

20.2.6 读写锁的应用

JDK1.5新的并发编程工具包中新增了读写锁，它是个轻量级、细粒度的锁，合理地使用读写锁，相比于传统的同步锁，可以提升并发访问的性能和吞吐量，在读多写少的场景下，使用同步锁比同步块性能高一大截。

尽管在JDK1.6之后，随着JVM团队对JIT即时编译器的不断优化，同步块和读写锁的性能差距缩小了很多，但是，读写锁的应用依然非常广泛。

下面对Netty中的读写锁应用进行分析，让大家掌握读写锁的用法。打开HashedWheelTimer代码，读写锁定义如下。

当新增一个定时任务的时候使用了读锁（如图20-19），用于感知wheel的变化。由于读锁是共享锁，所以当有多个线程同时调用newTimeout时，并不会互斥，这样，就提升了并发读的性能。

图20-19 Read Lock的使用

获取并删除所有过期的任务时，由于要从迭代器中删除任务，所以使用了写锁，如图20-20所示。

图20-20 Write Lock的使用

现将读写锁的使用场景总结如下。

- 主要用于读多写少的场景，用来替代传统的同步锁，以提升并发访问性能。
- 读写锁是可重入、可降级的，一个线程获取读写锁后，可以继续递归获取；从写锁可以降级为读锁，以便快速释放锁资源。
- `ReentrantReadWriteLock`支持获取锁的公平策略，在某些特殊的应用场景下，可以提升并发访问的性能，同时兼顾线程等待公平性。
- 读写锁支持非阻塞的尝试获取锁，如果获取失败，直接返回`false`，而不是同步阻塞，这个功能在一些场景下非常有用。例如多个线程同步读写某个资源，当发生异常或者需要释放资源的时候，由哪个线程释放是个难题，因为某些资源不能重复释放或者重复执行，这样，可以通过`tryLock`方法尝试获取锁，如果拿不到，说明已经被其他线程占用，直接退出即可。
- 获取锁之后一定要释放锁，否则会发生锁溢出异常。通常的做法是通过`finally`块释放锁。如果是`tryLock`，获取锁成功才需要释放锁。

20.2.7 线程安全性文档说明

当一个类的方法或者成员变量被并发使用的时候，这个类的行为如何，是该类与其客户端程序建立约定的重要组成部分。如果没有在这个类的文档中描述其行为的并发情况，使用这个类的程序员不得不做出某种假设。如果这些假设是错误的，这个程序就缺少必要的同步保护，会导致意想不到的并发问题，这些问题通常都是隐蔽和调试困难的。如果同步过度，会导致意外的性能下降，无论是发生何种情况，缺少线程安全性的说明文档，都会令开发人员非常沮丧，他们会对这些类库的使用小心翼翼，提心吊胆。

在Netty中，对于一些关键的类库，给出了线程安全性的API DOC（图20-21），尽管Netty的线程安全性并不是非常完善，但是，相比于一些做的更糟糕的产品，它还是迈出了重要的一步。

图20-21 ChannelPipeline的线程安全性说明

由于ChannelPipeline的应用非常广泛，因此，在API中对它的线程安全性进行了详细的说明，这样，开发者在调用ChannelPipeline的API时，就不用再额外地考虑线程同步和并发问题了。

20.2.8 不要依赖线程优先级

当有多个线程同时运行的时候，由线程调度器来决定哪些线程运行、哪些等待以及线程切换的时间点，由于各个操作系统的线程调度器实现大相径庭，因此，依赖JDK自带的线程优先级来设置线程优先级策略的方法是错误和非平台可移植的。所以，在任何情况下，程序都不能依赖JDK自带的线程优先级来保证执行顺序、比例和策略。

Netty中默认的线程工厂实现类，开放了包含设置线程优先级字段的构造函数。这是个错误的决定，对于使用者来说，既然JDK类库提供了优先级字段，就会本能地认为它被正确地执行，但实际上JDK的线程优先级是无法跨平台正确运行的。图20-22提供了一个线程优先级的反面示例。

图20-22 线程优先级的反面示例

20.3 总结

本章首先介绍了Java内存模型和多线程编程的基础知识，然后结合Netty的源码分析学习常用的多线程编程方法和技巧。

通过本章节的讲解，希望读者可以学以致用，在后续的工作中恰到好处地使用Java并发编程技术，提高系统的并发处理能力，提升产品的性能。

第21章 Netty架构剖析

本章将重点分析Netty的逻辑架构，通过对其关键架构质量属性的分析，让读者朋友能够更加深入地了解Netty的设计精髓。

希望读者在今后的架构设计中能够从Netty架构中汲取营养，设计出高性能、高可靠性和可扩展的产品。

本章主要包括：

- Netty逻辑架构分析
- 关键架构质量属性

21.1 Netty逻辑架构

Netty采用了典型的三层网络架构进行设计和开发，逻辑架构如图21-1所示。

图21-1 Netty逻辑架构图

21.1.1 Reactor通信调度层

它由一系列辅助类完成，包括Reactor线程NioEventLoop及其父类、NioSocketChannel/ NioServerSocketChannel及其父类、ByteBuffer以及由其衍生出来的各种Buffer、Unsafe以及其衍生出的各种内部类等。该层的主要职责就是监听网络的读写和连接操作，负责将网络层的数据读取到内存缓冲区中，然后触发各种网络事件，例如连接创建、连接激活、读事件、写事件等，将这些事件触发到PipeLine中，由PipeLine管理的职责链来进行后续的处理。

21.1.2 职责链ChannelPipeline

它负责事件在职责链中的有序传播，同时负责动态地编排职责链。职责链可以选择监听和处理自己关心的事件，它可以拦截处理和向后/向前传播事件。不同应用的Handler节点的功能也不同，通常情况下，往往会开发编解码Handler用于消息的编解码，它可以将外部的协议消息转换成内部的POJO对象，这样上层业务则只需要关心处理业务逻辑即可，不需要感知底层的协议差异和线程模型差异，实现了架构层面的分层隔离。

21.1.3 业务逻辑编排层（Service ChannelHandler）

业务逻辑编排层通常有两类：一类是纯粹的业务逻辑编排，还有一类是其他的应用层协议插件，用于特定协议相关的会话和链路管理。例如CMPP协议，用于管理和中国移动短信系统的对接。

架构的不同层面，需要关心和处理的对象都不同，通常情况下，对于业务开发者，只需要关心职责链的拦截和业务Handler的编排，因为应用层协议栈往往是开发一次，到处运行，实际上对于业务开发者来说，只需要关心服务层的业务逻辑开发即可。各种应用协议以插件的形式提供，只有协议开发人员需要关注协议插件，对于其他业务开发人员来说，只需关心业务逻辑定制即可。这种分层的架构设计理念实现了NIO框架各层之间的解耦，便于上层业务协议栈的开发和业务逻辑的定制。

正是由于Netty的分层架构设计非常合理，基于Netty的各种应用服务器和协议栈开发才能够如雨后春笋般得到快速发展。

21.2 关键架构质量属性

21.2.1 高性能

影响最终产品的性能因素非常多，其中软件因素如下。

- 架构不合理导致的性能问题。
- 编码实现不合理导致的性能问题，例如锁的不恰当使用导致性能瓶颈。

硬件因素如下。

- 服务器硬件配置太低导致的性能问题。
- 带宽、磁盘的IOPS等限制导致的I/O操作性能差。
- 测试环境被共用导致被测试的软件产品受到影响。

尽管影响产品性能的因素非常多，但是架构的性能模型合理与否对性能的影响非常大。如果一个产品的架构设计得不好，无论开发如何努力，都很难开发出一个高性能、高可用的软件产品。

“性能是设计出来的，而不是测试出来的”。下面我们看Netty的架构设计是如何实现高性能的。

（1）采用异步非阻塞的I/O类库，基于Reactor模式实现，解决了传统同步阻塞I/O模式下一个服务端无法平滑地处理线性增长的客户端的问题。

（2）TCP接收和发送缓冲区使用直接内存代替堆内存，避免了内存复制，提升了I/O读取和写入的性能。

（3）支持通过内存池的方式循环利用ByteBuf，避免了频繁创建和销毁ByteBuf带来的性能损耗。

（4）可配置的I/O线程数、TCP参数等，为不同的用户场景提供定制化的调优参数，满足不同的性能场景。

（5）采用环形数组缓冲区实现无锁化并发编程，代替传统的线程安全容器或者锁。

（6）合理地使用线程安全容器、原子类等，提升系统的并发处理能力。

（7）关键资源的处理使用单线程串行化的方式，避免多线程并发访问带来的锁竞争和额外的CPU资源消耗问题。

（8）通过引用计数器及时地申请释放不再被引用的对象，细粒度的内存管理降低了GC的频率，减少了频繁GC带来的时延增大和CPU损耗。

无论是Netty的官方性能测试数据，还是携带业务实际场景的性能测试，Netty在各个NIO框架中综合性能是最高的。下面，我们来看Netty官方的性能测试数据，如图21-2、21-3、21-4和21-5所示。

图21-2 64和128字节测试消息（本机网卡回环）

图21-3 256和1K字节测试消息（本机网卡回环）

图21-4 64和128字节测试消息（跨主机通信）

图21-5 256和1K字节测试消息（跨主机通信）

21.2.2 可靠性

作为一个高性能的异步通信框架，架构的可靠性是大家选择的一个重要依据。下面我们探讨Netty架构的可靠性设计。

1. 链路有效性检测

由于长连接不需要每次发送消息都创建链路，也不需要消息交互完成时关闭链路，因此相对于短连接性能更高。对于长连接，一旦链路建立成功便一直维系双方之间的链路，直到系统退出。

为了保证长连接的链路有效性，往往需要通过心跳机制周期性地链路检测。使用周期性心跳的原因是：在系统空闲时，例如凌晨，往往没有业务消息。如果此时链路被防火墙Hang住，或者遭遇网络闪断、网络单通等，通信双方无法识别出这类链路异常。等到第二天业务高峰期到来时，瞬间的海量业务冲击会导致消息积压无法发送给对方，由于链路的重建需要时间，这期间业务会大量失败（集群或者分布式组网情况会好一些）。为了解决这个问题，需要周期性的心跳对链路进行有效性检测，一旦发生问题，可以及时关闭链路，重建TCP连接。

当有业务消息时，无须心跳检测，可以由业务消息进行链路可用性检测。所以心跳消息往往是在链路空闲时进行发送的。

为了支持心跳，Netty提供了如下两种链路空闲检测机制。

- 读空闲超时机制：当连续周期T没有消息可读时，触发超时Handler，用户可以基于读空闲超时发送心跳消息，进行链路检测；如果连续N个周期仍然没有读取到心跳消息，可以主动关闭链路。

- 写空闲超时机制：当连续周期T没有消息要发送时，触发超时Handler，用户可以基于写空闲超时发送心跳消息，进行链路检测；如果连续N个周期仍然没有接收到对方的心跳消息，可以主动关闭链路。

为了满足不同用户场景的心跳定制，Netty提供了空闲状态检测事件通知机制，用户可以订阅空闲超时事件、写空闲超时事件、读或者写超时事件，在接收到对应的空闲事件之后，灵活地进行定制。

2. 内存保护机制

Netty提供多种机制对内存进行保护，包括以下几个方面。

- 通过对象引用计数器对Netty的ByteBuf等内置对象进行细粒度的内存申请和释放，对非法的对象引用进行检测和保护。
- 通过内存池来重用ByteBuf，节省内存。
- 可设置的内存容量上限，包括ByteBuf、线程池线程数等。

AbstractReferenceCountedByteBuf的内存管理方法实现如图21-6、21-7所示。

图21-6 对象引用

图21-7 对象引用释放

ByteBuf的解码保护，防止非法码流导致内存溢出，代码如图21-8所示。

图21-8 解码器单条消息最大长度上限保护

如果长度解码器没有单个消息最大报文长度限制，当解码错位或者读取到畸形码流时，长度值可能是个超大整数值，例如4294967296，这很容易导致内存溢出。如果有上限保护，例如单条消息最大不允许超过10M，当读取到非法消息长度4294967296后，直接抛出解码异常，这样就避免了大内存的分配。

3. 优雅停机

相比于Netty的早期版本，Netty5.0版本的优雅退出功能做得更加完善。优雅停机功能指的是当系统退出时，JVM通过注册的Shutdown Hook拦截到退出信号量，然后执行退出操作，释放相关模块的资源占用，将缓冲区的消息处理完成或者清空，将待刷新的数据持久化到磁盘或者数据库中，等到资源回收和缓冲区消息处理完成之后，再退出。

优雅停机往往需要设置个最大超时时间T，如果达到T后系统仍然没有退出，则通过Kill - 9 pid 强杀当前的进程。

Netty所有涉及到资源回收和释放的地方都增加了优雅退出的方法，它们的相关接口如表21-1所示。

表21-1 Netty重要资源的优雅退出方法

21.2.3 可定制性

Netty的可定制性主要体现在以下几点。

- 责任链模式：ChannelPipeline基于责任链模式开发，便于业务逻辑的拦截、定制和扩展。
- 基于接口的开发：关键的类库都提供了接口或者抽象类，如果

Netty自身的实现无法满足用户的需求，可以由用户自定义实现相关接口。

- 提供了大量工厂类，通过重载这些工厂类可以按需创建出用户实现的对象。
- 提供了大量的系统参数供用户按需设置，增强系统的场景定制性。

21.2.4 可扩展性

基于Netty的基础NIO框架，可以方便地进行应用层协议定制，例如HTTP协议栈、Thrift协议栈、FTP协议栈等。这些扩展不需要修改Netty的源码，直接基于Netty的二进制类库即可实现协议的扩展和定制。

目前，业界存在大量的基于Netty框架开发的协议，例如基于Netty的HTTP协议、Dubbo协议、RocketMQ内部私有协议等。

21.3 总结

本章首先对Netty的逻辑架构进行了分层和介绍，让读者能够从架构的层面了解Netty，随后对Netty的关键架构质量属性进行了详细分析和介绍。通过对Netty架构的剖析和讲解，希望读者能够掌握如何设计高性能、高可靠性、可定制性和可扩展的软件架构。

第22章 Netty行业应用

随着移动互联网的发展，使用轻量级、分布式、具备弹性伸缩能力的架构代替传统基于Tomcat等Web容器的垂直架构已经成为一种必然的趋势。

服务拆分和分布式部署之后，各服务节点之间需要通过通信协议进行跨节点通信，考虑到是内部节点之间的通信，从性能和可维护性角度考虑，往往会选用性能更高、扩展性更好的内部私有协议。

随着大数据的应用和发展，基于Hadoop的MapReduce得到了越来越多的应用。Hadoop中的MapReduce是一个使用简易的软件框架，基于它写出来的应用程序能够运行在由上千个商用机器组成的大型集群上，并以一种可靠容错的方式并行处理上T级别的数据集。

无论是互联网的分布式服务框架，还是Hadoop的并行计算框架，它们集群组网中的各个节点需要通过高性能的通信框架来实现同步或者异步RPC调用。Netty作为成熟高性能的异步通信框架，成为了这些产品的首选。

本章主要包括：

- Netty在互联网行业的应用
- Netty在大数据领域的应用
- Netty在网络游戏服务器中的应用

22.1 Netty在互联网行业的应用

22.1.1 传统垂直架构面临的问题

随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构势在必行，亟需一个治理系统来确保架构有条不紊地演进。互联网架构的演进历史如图22-1所示。

图22-1 互联网架构的演进历史

使用传统垂直应用架构面临的主要挑战如下。

（1）前后台耦合，业务无法有效拆分，随着规模的膨胀和业务越来越复杂，系统的开发和维护成本越来越高，最终会导致不可控。

（2）由于缺乏高性能的RPC框架，导致集群节点间通信效率低下，系统无法平滑扩容。

（3）缺少统一的服务注册中心对集群服务进行管理，导致系统无法弹性伸缩。

（4）当服务越来越多时，无法对服务进行有效的容量评估和服务治理。

22.1.2 阿里分布式服务框架Dubbo

作为分布式服务框架，使用Dubbo将传统基于Tomcat等垂直本地应用进行服务化。分布式服务化面临的主要问题如下。

（1）服务分布式部署之后，需要高性能的内部通信协议实现异步

RPC调用。

(2) 当服务越来越多时，服务URL配置管理变得非常困难，F5硬件负载均衡器的单点压力也越来越大。

(3) 当进一步发展后，服务间依赖关系变得错综复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。

(4) 服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？

Dubbo的解决方案如下。

(1) 基于Netty+二进制编解码框架实现了内部私有协议——Dubbo协议，来代替原来的RMI、HTTP+XML等协议，提升节点之间的通信性能。

(2) 将Zookeeper作为服务注册中心，动态地注册和发现服务，使服务的位置透明，并通过在消费方获取服务提供方地址列表，实现软负载均衡和Failover，降低对F5硬件负载均衡器的依赖，也能减少部分成本。

(3) 服务治理框架自动画出应用间的依赖关系图，以帮助架构师理清关系。

(4) 为了解决这些问题，首先，要将服务现在每天的调用量和响应时间都统计出来，作为容量规划的参考指标。其次，要可以动态调整权重，在线上将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阈值，记录此时的访问量，再以此访问

量乘以机器数反推总容量。

Dubbo的服务治理框架如图22-2所示。

图22-2 Dubbo服务治理框架

22.1.3 Dubbo的架构介绍

Dubbo的注册中心基于Zookeeper实现，服务的订阅和发布流程如图22-3所示。

图22-3 Dubbo服务订阅和发布流程

各系统节点的角色说明如下。

- **Provider**: 暴露服务的服务提供方。
- **Consumer**: 调用远程服务的服务消费方。
- **Registry**: 服务注册与发现的注册中心。
- **Monitor**: 统计服务的调用次数和调用时间的监控中心。
- **Container**: 服务运行容器。

各节点调用关系说明如下。

- 0. 服务容器负责启动、加载、运行服务提供者。
- 1. 服务提供者在启动时，向注册中心注册自己提供的服务。
- 2. 服务消费者在启动时，向注册中心订阅自己所需的服务。
- 3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
- 4. 服务消费者从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

- 5. 服务消费者和提供者在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

Dubbo架构的主要质量属性如下。

（1）连通性

- 注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小。
- 监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示。
- 服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销。
- 服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销。
- 注册中心、服务提供者、服务消费者三者之间均为长连接，监控中心除外。
- 注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者。
- 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表。
- 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

（2）健壮性

- 监控中心宕掉不影响使用，只是丢失部分采样数据。
- 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务。
- 注册中心对等集群，任意一台宕掉后，将自动切换到另一台。
- 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存进行通信。
- 服务提供者无状态，任意一台宕掉后，不影响使用。
- 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复。

（3）伸缩性

- 注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心。
- 服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者。

（4）升级性

- 当服务集群规模进一步扩大，带动IT治理结构进一步升级，需要实现动态部署，动态部署的流程如图22-4所示。

图22-4 Dubbo动态发布服务和升级

22.1.4 Netty在Dubbo中的应用

Dubbo的RPC远程服务调用默认使用Netty+Dubbo协议实现，传统RPC服务调用的问题如下。

- 网络传输方式使用同步阻塞I/O（BIO）。
- 序列化方式使用Java或者JSON等，性能差，序列化后的码流太大，浪费带宽。
- 传统的同步阻塞I/O每个TCP连接占用一个线程。
- JDK动态代理的性能太差。

高性能的RPC框架需要解决的问题有以下三个。

- 协议：用什么数据格式进行传输，通信双方的契约。
- 传输：用什么样的通道将数据发送给对方。
- 线程：当接收到数据时，如何分发数据进行处理。

Dubbo RPC框架默认推荐使用Dubbo协议进行通信和数据传输，相比于旧的Hessian协议其性能更高，而且支持异步I/O通信。

Dubbo的远程服务调用数据流程图如图22-5所示。

图22-5 Dubbo远程服务调用

Dubbo的RPC框架通过Dubbo encode方法将POJO对象编码为Dubbo协议的二进制字节流，通过Netty Client发送给服务提供者，服务提供者的Netty Server从NioSocketChannel读取二进制码流，将ByteBuffer解码为Dubbo请求消息并调用服务提供者，服务提供者处理完成之后，构造应答，通过RPC框架返回给服务调用者。Dubbo协议消息头定义如图22-6所示。

图22-6 Dubbo协议消息头定义

22.1.5 Dubbo框架集成Netty源码分析

Netty作为Dubbo RPC框架的高性能异步NIO框架，它的主要职责如下。

- (1) 提供异步、高性能的NIO通信框架。
- (2) NIO客户端和服务端。
- (3) 心跳检测能力。
- (4) 断连重连机制。
- (5) 流量控制。
- (6) Dubbo协议的编解码Handler。

下面我们看它的源码实现，Netty框架的代码被封装在transport的netty包下，源码的目录结构如图22-7所示。

图22-7 Netty封装相关代码

由于承载Dubbo协议的是TCP而不是具体的某个NIO框架，所以Dubbo的RPC自身不能与Netty框架耦合。一旦耦合，未来升级Netty的版本或者切换其他的NIO框架都会带来接口层面的不兼容，Dubbo协议本身也同时支持Mina和grizzly，所以，它必须对NIO框架进行统一地抽象，通过继承或者聚合的方式对底层NIO框架的实现细节进行屏蔽。

首先看NettyClient，它负责绑定本地端口，发起连接，关闭连接，进行重连等，相关代码如图22-8所示。

图22-8 Netty客户端启动

Dubbo Netty客户端的封装与之前讲解的其他例子类似。

(1) 向ChannelPipeline中新增Dubbo协议解码器。

(2) 向ChannelPipeline中新增Dubbo协议编码器。

(3) 向ChannelPipeline中新增Dubbo Handler。

下面我们继续看Dubbo实现的Netty Handler，它的相关方法如图22-9所示。

图22-9 Dubbo协议的ChannelHandler

NettyHandler持有了Dubbo自定义的ChannelHandler，通过ChannelHandler回调Dubbo的Filter，实现RPC服务调用，图22-10为示意图。

图22-10 Dubbo基于Filter的服务调用

Dubbo的Netty服务端职责如下。

- 绑定本地监听端口，接收客户端的连接。
- 管理客户端的连接状态。

它的代码实现如图22-11所示。

图22-11 Dubbo Netty服务端实现

由于实现原理与前几章介绍的demo类似，大家可以举一反三，自行阅读相关代码，此处不再赘述。

心跳检测机制：Dubbo没有直接使用Netty的链路空闲检测机制进行心跳功能的开发，而是提供了独立的ScheduledThreadPoolExecutor、HeartBeatTask和HeartbeatHandler来实现心跳检测功能。

启动心跳的代码如图22-12所示。

图22-12 启动Dubbo心跳

停止心跳的代码如图22-13所示。

图22-13 停止Dubbo心跳

提供停止心跳检测方法的原因是：当链路中断，在客户端重连成功之前，是不需要发送心跳的，由于心跳是无效循环定时器，所以链路中断期间需要停止心跳检测，否则会导致功能异常。

心跳检测Task的代码实现如图22-14所示。

图22-14 Dubbo的心跳检测Task

代码很简单，就是构造心跳请求消息，通过Channel进行发送。心跳检测超时，需要关闭链路，代码如图22-15所示。

图22-15 Dubbo的心跳检测超时处理

22.2 Netty在大数据领域的应用

Apache Avro是一个独立于编程语言的数据序列化系统，它同时提供了RPC服务调用能力。该项目是由Hadoop之父Doug Cutting创建的，旨在解决Hadoop中Writable类型的不足——缺乏语言的可移植性。拥有一个可被多语言处理的数据格式与绑定到单一语言的数据格式相比，前者更易于与公众共享数据集。允许其他编程语言能够读写序列化的数据会使架构具有更好的扩展性和移植性。

Avro完全依赖模式（Schema），通过Schema定义各种数据结构，只有确定了Schema才能对数据进行解释，所以在数据的序列化和反序列化之前，必须先确定Schema的结构。正是Schema的引入，使得数据具有了自描述的功能，同时能够实现动态加载，另外与其他的数据序列化系统如Thrift相比，数据之间不存在另外的任何标识，有利于提高数据处理的效率。Avro的诸多优势，使其将成为代替Hadoop现有RPC框架的下一代通信中间件。

Avro的RCP服务支持以下两种模式。

- 传统基于Jetty的HTTP Server；
- 新的基于Netty的Netty Server。

Avro属于非常轻量级，实现简洁。在使用方面便于使用者进行二次开发，它的逻辑架构分为两层。

（1）网络传输层使用Netty的异步通信实现（也支持HTTP传输）；

(2) 协议层可扩展，目前支持的数据序列化方式有Avro，Protocol Buffer，JSON，Hessian，Java序列化，使用者可以注册自己的协议格式及序列化方式以实现协议的自定义扩展。

Avro 框架的主要特点如下。

(1) 客户端传输层与应用层逻辑分离，传输层主要职责包括创建连接、连接查找与复用、传输数据、接收服务端回复后回调应用层。

(2) 客户端支持同步调用和异步调用，服务调用异步化能很好地提高系统吞吐量，降低对方应答缓慢或者超时导致的服务线程被Hang住，为防止异步发送请求过快，客户端增加了“请求流量限制”功能。

(3) 服务端有一个协议注册工厂和序列化注册工厂，这样便于针对不同的应用场景来定制远程服务调用方式。RPC框架只是远程服务调用的一种实现方式。在分布式的系统架构中，分布式节点之间的通信会存在多种方式，比如MQ的TOP消息，一个消息可以有多个订阅者，也可以通过发送事件的方式进行异构系统或者分布式节点直接的服务调用。

(4) Avro序列化框架是Hadoop下的一个子项目，其特点是数据序列化不带标签，因此序列化后的数据非常小。支持动态解析，不像Thrift 与 Protocol Buffer必须根据IDL来生成代码，这样对用户代码侵入性比较强。

(5) 序列化性能高，基本上能够和 Protocol Buffer持平，远远高于Java序列化、JSON序列化等。

Netty在Avro RPC框架中的职责与Dubbo中的类似，提供高性能的异

步NIO通信能力，由于代码实现原理类似，所以此处不再赘述，感兴趣的读者可以从Avro的官网（<http://avro.apache.org/>）下载源码进行分析和学习。

22.3 Netty在游戏行业的应用

MMORPG服务端架构设计目标如下。

- 廉价的系统资源配置，例如商用PC机组成的集群系统。
- 低学习成本，例如使用Java和JS替换C++和PHP。
- 高性能。
- 高可靠性。
- 可扩展。
- 可跨平台运行。

要达到上述目标并不是件容易的事情，下面我们简单学习一下游戏服务器的架构以及如何在游戏服务端使用Netty。

22.3.1 游戏服务端架构介绍

通常情况下，一个大型的游戏服务端的结构如图22-16所示。

图22-16 游戏服务器架构

玩家通过账户和密码进行登录，流程如图22-17所示。

图22-17 玩家登录认证流程

登录认证流程如下。

- ①使用账户、密码，发出请求。
- ②黑名单、掩码过滤。

③允许/拒绝。

④账户、密码有效。

⑤允许/拒绝。

⑥是否有账户余额。

⑦允许/拒绝。

⑧加密用户ID，生成令牌。

⑨允许/拒绝，最终返回应答。

选择玩家的流程如图22-18所示，说明如下。

图22-18 选择玩家流程图

①使用令牌，发出请求。

②空索引队列。

③是否存在空位。

④请求账户角色列表。

⑤获取角色列表。

⑥向玩家反馈角色信息。

⑦玩家选择角色。

⑧玩家ID信息、玩家位置信息等。

- ⑨根据规则，提议地图服务器。
- ⑩加密用户ID与地图服务器ID，生成令牌。
- ⑪选择地图服务器，最后返回应答。

进入游戏流程图（图22-19）。

图22-19 进入游戏流程图

- ①使用令牌，发出请求。
- ②用户ID是否在角色列表。
- ③是则处理，否则等待。
- ④地图服务器处理或等待。
- ⑤客户端随机请求。
- ⑥服务器端随机回复。
- ⑦其他玩家信息间歇性请求。
- ⑧其他玩家信息间歇性回复。

游戏服务器集群组网示意图（图22-20）。

图22-20 游戏服务器集群组网

22.3.2 Netty在游戏服务端的应用

Netty在游戏服务端的应用，主要体现在以下几点。

（1）游戏服务器有多种角色，它们之间需要频繁地通信，例如地图服务器、网关服务器、聊天服务器等。利用Netty的异步NIO框架，可以为各进程之间提供高性能的异步网络通信能力。

（2）灵活的编解码定制能力，可以满足不同游戏场景下多协议和私有协议编解码。

（3）可配置的线程池、TCP参数等可以为不同的游戏服务器进程提供差异化的定制能力。

（4）SSL、黑白名单过滤等可以直接用于登录认证等流程。

（5）心跳检测、流量整形、日志统计等原生的能力，可以提升游戏服务器的可服务性。

（6）基于内存池的对象重用技术，更大程度上重用对象，节省内存，降低GC的频度，降低玩家被卡的概率。

使用Netty作为游戏服务器之间的内部通信组件之后的组网图如图22-21所示。

图22-21 用Netty作为游戏服务器之间的内部通信组件之后的组网图

22.4 总结

Netty作为一个成熟的异步NIO通信框架，内置了多种序列化类库，自身也提供了很多附加的功能。我们发现，无论是互联网的分布式服务框架，大数据的RPC、序列化框架，还是游戏服务器。尽管业务场景差异很大，但是它们对Netty的使用原理基本一致，都是利用它提供的异步通信能力进行跨节点的数据传输或者服务调用。

无论场景如何改变，只要涉及到系统的集群组网或者分布式部署，就需要RPC调用能力，由于Netty的优异表现，它往往是首选的异步通信框架。希望通过本章的学习，读者可以在实际工作中灵活地使用Netty为自己的产品服务。

第23章 Netty未来展望

作为本书的结尾章节，和读者朋友们一起展望下Netty的未来。

本章主要包括：

- 应用范围
- 技术演进
- 社区活跃度
- Road Map

23.1 应用范围

随着大数据、互联网和云计算的发展，传统的垂直架构逐渐将被分布式、弹性伸缩的新架构替代。

系统只要分布式部署，就存在多个节点之间通信的问题，由于是内部通信，同时强调高可扩展性和高性能，因此往往会选择高性能的通信方式，利用Netty + 二进制编解码承载这些内部私有协议，已经逐渐成为业界主流的用法。例如阿里的分布式服务框架Dubbo、RocketMQ、Hadoop的Avro等。

随着JDK7的逐渐普及，Java的原生NIO类库已经升级到了NIO2.0，未来越来越多基于传统Socket编程的应用程序会切换到新的NIO类库上，考虑到切换和维护成本，大多数公司将会选择Netty或者Mina作为高性能的NIO框架来实现异步通信。

可以预见在未来2~3年内，基于NIO的异步通信将成为Java网络编程的主流，未来Netty的应用范围将会越来越广。

23.2 技术演进

随着JDK8的推出，ORACLE公司也加大了JDK7的推广力度，并给出了JDK6的deadline。Netty的5.X系列版本将紧跟JDK的发展潮流，可以预测，越来越多JDK7的新特性将被Netty 5.X系列版本使用，最引人注目的一个就是NIO2.0类库中AIO的使用。

让我们拭目以待Netty 5.0正式版本的推出吧。

23.3 社区活跃度

Netty的社区一直非常活跃，API文档和开发指南内容也比较全面，Bug的修复速度相对较快，这些因素促进了Netty社区的良性发展。

23.4 Road Map

Netty的版本更新节奏非常快，主要原因如下。

- Bug的修正速度较快。
- 新特性的推出速度快。

下面我们一起看下Netty 4.X系列的版本更新情况，如图23-1所示。

图23-1 Netty 4.X系列的版本更新一览表

2013年12月22日，Netty推出了新的Netty5.0.0.Alpha1，这预示着2014年Netty将会不断推出5.0系列的公测和正式版本，可以预测，5.0系列的第一个Final版本可能会在2014年底推出，届时，“赶时髦”的读者朋友们就可以考虑是否使用和升级最新的Netty5.0系列版本，体验更多的新特性和新功能。

23.5 总结

作为本书的最后一个章节，我们一起展望了Netty的美好未来，作为最有影响力的NIO框架，Netty得到了众多架构师和程序员的喜爱。希望在未来的工作中，读者能够把Netty用起来，用好它，让它为你的项目、你的公司创造更大的价值。

附录 **Netty**参数配置表
